MAIL ROOM
JUL
12
1996
38
PAT. & TRADE MARK OFF.

PATENT
38600.0001(B)

TURING COMPLETE COMPUTER IMPLEMENTED MACHINE LEARNING METHOD AND SYSTEM

BY

Peter Nordin
Wolfgang Banzhaf

"Express Mail" Receipt No. EF740909751US

Date of Deposit July 12, 1996
I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 on the date indicated above and is addressed to the Commissioner of Patents and Trademarks, Washington, D.C. 20231.

Tara K. Inouye-Hill
Name of Person Mailing

[illegible]
Signature of Person Mailing

43

38600-0001

MAIL ROOM JUL 12 1996 38 PATENT & TRADEMARK

08/682859

A No Fee

TURING COMPLETE COMPUTER IMPLEMENTED MACHINE LEARNING METHOD AND SYSTEM

BACKGROUND OF THE INVENTION

Field of the Invention

The present invention generally relates to the art of computerized computation systems for performing repeated computations on data that is not known until a computer user is running the system ("run-time", and "run-time data") and more specifically to a system of creating, initializing, storing, altering and executing both the run-time data and the computer code necessary to execute the repeated computations (the "related code") in native machine code operating on a register machine. The present invention relates also to the art of computerized learning systems (which are usually characterized by the need to perform repeated computations on run-time data) and more specifically to a register machine learning method in which the information and/or the computer program(s) that constitute solutions to a problem are created, initialized, stored, altered and executed in native machine code by using a higher level programming language to produce an optimized solution through the direct application of learning algorithms to the information stored in the native machine code.

Description of the Related Art

Machine learning systems have been proposed in the art for the solution of problems such as classification, prediction of time-series data, symbolic regression, optimal control, etc. Examples of various machine learning systems are neural networks, fuzzy networks, genetic algorithms (including genetic programming and classifier systems), Evolutionary Strategies, Evolutionary Programming, ADATE program induction, cellular automata, Box Jenkins optimization, ARMA optimization and many others. Rather than applying a direct computational approach, these systems create one or more proposed solutions in the form of data and "computer program" entities, and iteratively alter

2

the data and/or entities for the purpose of finding an optimal solution to a problem. One such approach is described in, for example, U.S. Patent No. 4,935,877, entitled "NON-LINEAR GENETIC ALGORITHMS FOR SOLVING PROBLEMS", issued June 19, 1990 to John Koza.

The set of practically solvable problems is highly related to the efficiency of the algorithm and implementation. It is therefore important to minimize the overhead involved in executing any machine learning system.

Machine learning systems create learned solutions to problems. These solutions have two elements: (1) Elements that hold the information that is learned during the execution of the machine learning system (the "Learned Elements"); and (2) Elements that are necessary to convert the Learned Elements into meaningful action or output by the computer (the "Conversion Elements"). Existing machine learning approaches other than the present invention can be classified into two categories in terms of how they store and execute both the Learned Elements and the Conversion Elements. Those two categories are compiler based systems and interpreted systems.

An interpreted system is written in a high level language such as LISP and both the Learned and Conversion Elements are held in data-like structures such as LISP lists, which are converted ("interpreted") into native machine code at run-time by an interpreter. The interpreter, itself, also contains Conversion Elements for such interpreted systems. So for example, U.S. Patent No. 4,935,877 uses, as its Learning Elements various high level LISP expressions, customized for the problem at hand, to represent, symbolically, a "computer program." That is, a high level "program" structure symbolized by the LISP List, is itself the subject of learning in that system. In this system, the Learned Elements are represented as a hierarchical tree structure. This solution gives good flexibility and the ability to customize the language depending on the constraints of the problem at hand.

The principal disadvantage of this interpreting approach to

3

machine learning is that the Learned Elements and many of the Conversion Elements are stored in high level, symbolic data-like structures. Computers can operate only by executing native machine code. Thus, interpreting machine learning systems learn by modifying high level symbolic representations (the Learned Elements) that are, ultimately, converted into machine code by the interpreter at run-time. The need to convert (interpret) the Learned Elements and some of the Conversion Elements into native machine code at run-time before any useful action or output may be had from a computer is very time consuming and involves a large amount of overhead in machine resources such as CPU time, RAM memory, and disk space. In effect, all of the Learned Elements and the Conversion Elements in the LISP List are treated as run-time data that must be accessed and converted to machine code before any useful action or output may be had. Simply put, interpreted systems are slow and use a lot of a computer system's resources.

Other machine learning systems are written in high level compiled computer languages such a C, C++, Pascal and so forth. This is the "compiler based" approach to machine learning. Large amounts of the Conversion Elements in such systems are compiled before run-time into native machine code in such a compiler based approach. Because those compiled Conversion Elements are already in native machine code at run-time, there is no need to interpret these Conversion Elements at run-time. Using a compiler based approach instead of an interpreted approach usually results in a substantial speed-up of the execution of the machine learning system, often increasing the speed of learning by a factor of ten times. Examples of such compiler based systems are GPC by Walter Tackett (genetic programming), Lil-GP (genetic programming), and EvoC (evolutionary strategies).

The compiler based approach to machine learning, while faster than the interpreted approach, must still access run-time data that is stored in data structures held in RAM memory or in some other form of memory such as hard disk. The reason that

4

run-time data structures must be accessed in compiler based machine learning systems (or any machine learning system other than the present invention) is that the process of learning involves initializing and then altering the Learned Elements at run-time.

For example, the weights in a neural network are Learned Elements for neural network applications. Compiler based neural network systems hold those weights in data structures such as arrays or linked lists. Similarly, compiler based genetic programming systems store symbolic representations of program structures (the Learned Elements in genetic programming) in RAM data structures such as arrays, linked lists or parse trees. In all compiler based machine learning systems, the already compiled Conversion Elements must repeatedly access the Learned Elements (weights or symbolic representations of program structures) from the data structures in RAM memory in order to execute and evaluate the Learned Elements and to modify the Learned Elements according to the learning algorithm that is being used to modify the Learned Elements during learning. Such repeated access is necessary before any meaningful output or action may be had from a computer based on the Learned Elements. Such repeated accesses to RAM data structures is time consuming and uses extensive amounts of RAM to hold the Learned Elements.

More generally, computing systems that perform repeated calculations on run-time data may also be categorized as compiler based systems or interpreted systems. They store access, alter and execute run-time data in a manner similar to the storage, access, alteration and execution of the Learned Elements in the systems described above and are subject to the same limitations of slow execution and system overhead as the systems described above. By the phrase, "repeated calculations (or computations) on (or of) run-time data," this application means the execution of one or more instructions that must access one or more elements of run-time data (from data storage such as RAM or hard disk) more than once on the same values of the run-time data.

5

SUMMARY OF THE INVENTION

The present invention utilizes the lowest level binary machine code as the "entities" or individuals or solutions. Every individual is a piece of machine code that is called and manipulated by the genetic operators.

There is no intermediate language or interpreting part of the program. The machine code program segments are invoked with a standard C function call. Legal and valid C-functions are put together, at run time, directly in memory by the genetic algorithm. The present system can therefore be considered as a "compiling" machine learning implementation.

The present system generates binary code directly from an example set, and there are no interpreting steps. The invention uses the real machine instead of a virtual machine, and any loss in flexibility will be well compensated for by increased efficiency.

More specifically, one or more machine code entities such as functions are created which represent (1) solutions to a problem or (2) code that will perform repeated calculations on "run-time data" that is encapsulated into the machine code. These entities are directly executable by a computer. The programs are created and altered by a program in a higher level language such as "C" which is not directly executable, but requires translation into executable machine code through compilation, interpretation, translation, etc.

The entities are initially created as an integer array that can be altered by the program as data, and are executed by the program by recasting a pointer to the array as a function type. The entities are evaluated by executing them with training data (as defined elsewhere) as inputs, and calculating "fitnesses" based on a predetermined criterion or by recovering the output as the result of one of the repeated calculations on run-time data.

After one or more "executions," the entities are then altered by recasting the pointer to the array as a data (e.g. integer) type. Or the original data pointer to the array may have been typecast earlier to a function pointer in a way that did not permanently change the type of the pointer. In that case, the original data pointer is used in its original form. This process is iteratively repeated until an end criterion is reached.

In the case of machine learning, the entities change in such a manner as to improve their fitness, and one entity is ultimately produced which represents an optimal solution to the problem. In the case of repeated calculations on run-time data, the entity permits very rapid calculations to be performed generating usable output from the run-time data.

Each entity includes a plurality of directly executable machine code instructions, a header, a footer, and a return instruction. The alteration process is controlled such that only valid instructions are produced. The headers, footers and return instructions are protected from alteration.

The system can be implemented on an integrated circuit chip, with the entities stored in high speed memory in a central processing unit.

The present invention overcomes the drawbacks of the prior art by eliminating all compiling, interpreting or other steps that are required to convert a high level programming language instruction such as a LISP S-expression into machine code prior to execution.

Experiments have demonstrated that the present approach can speed up the execution of a machine learning or a repeated calculation system by 1,000 times or more as compared to systems which provide potential solutions in the form of high level "program" expressions in the interpreted approach. The speedup is in excess of 60 times over the compiler based approach. This makes possible the practical solutions to problems which could not heretofore be solved due to excessive computation time. For example, a solution to a difficult problem can be produced by the present system in hours, whereas a comparable solution might take years using conventional techniques.

These and other features and advantages of the present invention will be apparent to those skilled in the art from the following detailed description, taken together with the accompanying drawings, in which like reference numerals refer to like parts.

DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram illustrating a machine learning system according to the present invention as implemented on a general purpose computer;

FIG. 2 is a block diagram illustrating a computer program which implements the functionality of the present invention as stored in a random access memory of the computer of FIG. 1;

FIG. 3 is a block diagram illustrating banks of registers in the computer used in the preferred implementation of the invention;

FIG. 4 is a flowchart illustrating a method of machine learning according to the invention;

FIG. 5 is a diagram illustrating types of machine code instructions which are used by the preferred implementation of the invention;

FIG. 6 is a detailed flowchart illustrating a specific embodiment of the preferred implementation of the invention;

FIG. 7 is a diagram illustrating an array of functions that are used by the invention;

FIG. 8 is a diagram illustrating an alternative function that can be used by the invention;

FIG. 9 is a diagram illustrating the present machine learning system as implemented on a specially designed integrated circuit chip;

FIGs. 10a and 10b are diagrams illustrating a genetic uniform crossover operation;

FIGs. 11a and 11b are diagrams illustrating a genetic 2-point "swap" crossover operation;

FIGs. 12a and 12b are diagrams illustrating a genetic operator crossover operation;

FIGs. 13a and 13b are diagrams illustrating a genetic operand crossover operation;

FIG. 14 is a diagram illustrating a genetic operator mutation operation;

FIG. 15 is a diagram illustrating a genetic operand mutation operation;

FIGs. 16a to 16d in combination constitute a flowchart illustrating a generic implementation of the invention to machine learning;

FIGs. 17a to 17d in combination constitute a flowchart illustrating a generic implementation of the invention to repetitive computations based on run time data;

FIGs. 18a to 18h in combination constitute a flowchart illustrating an machine learning implementation of the system which induces machine code functions where the Learned Elements are machine code instruction structure and machine code instruction contents. This is one implementation of what is referred to loosely as Compiling Genetic Programming Systems;

FIGs. 19 to 21 are diagrams illustrating the use of functions and registers in accordance with the invention; and

FIGs. 22a to 22k are flowcharts illustrating a detailed implementation of the invention.

DETAILED DESCRIPTION OF THE INVENTION

The present method must be implemented using a computer due to the immense number of complex computations that must be made. The computer can be a general purpose computer, with the functionality of the method being implemented by software. Alternatively, as will be described below, part or all of the functionality of the system can be implemented on a specially designed integrated circuit chip.

Introduction

The present invention utilizes the lowest level native machine code with no immediate step of compilation or interpretation to store, access, initialize, create, alter and execute run-time data and related code where repeated computations (embodied in the related code) must be performed using that run-time data. In the context of machine learning, the run-time data (the Learned Elements) and the computations that are to be performed on the run time data (the Conversion Elements) are created, initialized, stored, altered and executed directly in native machine code with no intermediate step of compilation or interpretation. Thus, in the context of machine learning, the present invention stores both Learned Elements and the Conversion Elements in native machine code. All or many of the Learned Elements and all of the Conversion elements are created, initialized, stored, altered and executed directly in the native machine code with no intermediate step of compilation or interpretation.

The present invention is not limited to any particular machine learning paradigm such as, and by way of example only, genetic programming, genetic algorithms, simulated annealing or neural networks. Rather, it is a method of creating, initializing, storing, altering and executing all or part of the Learned Elements and the Conversion Elements in any machine learning paradigm directly in native machine code.

For example, when the present invention is applied to evolving native machine code structure and contents, it creates, initializes, stores, alters and executes the programs structures that are to be evolved (the Learned Elements) directly in native machine code. The approach of the present invention is completely unlike the compiler based and interpreting approaches to evolving "computer programs," which only create, store, alter and (with the aid of a compiler or an interpreter) execute high level symbolic representations of high level "program structures" in a high level programming language that, ultimately, represent and are converted into actual machine code by an interpreter or that are executed by compiled code making repeated accesses to the run-time Learned Elements (the representations of high level "program structures") in RAM.

As a further example, when the present invention is applied to neural networks, it creates, initializes, stores, alters and executes the weights of the network (the Learned Elements) and

10

the code that executes a network characterized by those weights (the Conversion Elements) directly in native machine code. This is completely unlike the compiler based and interpreting approaches to neural networks, which create, initialize, store, alter and (with the aid of a compiler or an interpreter) execute high level symbolic representations of weights and network structures that, ultimately, represent and are converted by an interpreter into the actual machine code that executes the neural network or that are executed by compiled code making repeated accesses to the run-time Learned Elements in RAM.

The present invention utilizes functions created in the lowest level machine code as the "learning entity" for machine learning or as the "computation entity" for repeated computations based on run-time data. Every such learning entity or computation entity (also referred to as an "entity" or an "individual" or a "solution") in the present invention is a complete, executable native machine code function that can be executed from a higher level language such as C or assembler by a simple standard function call ("Native Functions"). All of the run-time data and related code, the Learned Elements and the Conversion Elements are assembled by the present invention at run time into Native Functions, which constitute the learning entity or the computation entity, and the learning entity or computation entity are then stored, altered and executed directly as Native Functions. When the run-time data or the Learned Elements change, the learning entities or the computation entities must also be changed because the previous run-time data, including the previous Learned Elements, are already included in the Native Function learning entity. The present invention is, in part, a method of making such changes directly in the learning and computation entities. In a sense, the present system acts as an on-the-fly compiler of both real time data and related code (for computation entities), and of Learned Elements, and Conversion Elements (for learning entities). The present system can therefore be loosely considered as a "compiling" machine learning implementation. In the specific case of evolving machine code structures and contents as the Learned Elements, the present methodology can be loosely described as a "Compiling Genetic Programming System (CGPS)".

The present system generates low level machine code directly from an example set, and there are no interpreting or compiling steps. The invention uses the real machine instead of a virtual machine, and any loss in flexibility will be well compensated for by increased efficiency.

More specifically, one or more individuals (or "entities" or "solutions") are created which represent solutions to a problem and are directly executable by a computer as Native Functions. The programs are created, initialized, stored, altered and executed by a program in a higher level language such as "C" which is not directly executable, but requires translation into executable machine code through compilation, interpretation, translation, etc.

The entities are initially created as an integer array that can be created, initialized, stored and altered by the higher level program as data. Although the present implementation of the system uses arrays of integers, the system could be implemented as arrays of any other data type and such implementations are within the scope and spirit of the present invention.

To understand how the system chooses what integers are included in this array, one must understand what a low level machine instruction looks like. The low level instructions on all existing CPU's are groups of binary bits. On the Sun system for which most of the development of the present system has taken place, all instructions are 32 bits long. In low level machine code, each bit has a particular meaning to the CPU. In the present system, the integers chosen for the integer arrays that constitute the entities are integers that, when viewed in binary format, correspond to valid low level machine code instructions for the CPU used in the computer system to which the present invention is being applied. Although the present system is implemented on Sun and Intel systems, it could be implemented on any CPU and many other chips. Such alternative implementations are within the scope and spirit of the present invention.

In the present implementation of the invention in C, the entities are executed by typecasting a pointer to the array of integers that constitute the entity to be a pointer to a function. The function pointed to by the entity is then executed by calling the function using the function pointer like a regular C function. The important point here is that the entities are viewed, alternatively, as data and as functions at different times in the system. At the time the entities are created, initialized, stored and altered, the entities are viewed by the system as data-an array of integers. At the time the entities are executed, they are viewed as Native Functions. So if the present system was implemented in assembler, the entities would be created as integers (as data) and they would be executed as Native Functions. No typecast would be required in assembler but such an implementation is within the scope and spirit of the present invention.

When the entities are executed, they use training data (defined below) as inputs. The results of the execution are converted into some measure of how well the problem is solved by the entity, said measure being determined by the machine learning paradigm or algorithm being used. Said measure will be referred to herein as "fitness" regardless of the machine learning paradigm or algorithm being used. Many machine learning algorithms take the measure of fitness and feed it back into the algorithm. For example, evolutionary based learning algorithms use fitness as the basis for selecting entities in the population for genetic operations. On the other hand simulated annealing uses fitness, inter alia, to determine whether to accept or reject an entity as being the basis for further learning. After entities are initialized, most machine learning algorithms, go through repeated iterations of alteration of entities and execution of entities until some termination criterion is reached.

In the particular implementation of the present invention, the entities are initially created and initialized as an integer array that can be altered by the program as data and are executed by the program by recasting a pointer to the array as a pointer to a function type. The entities are evaluated by executing them with training data as inputs, and calculating fitnesses based on a predetermined criterion applied to the output of an entity after execution.

The entities are then altered based on their fitnesses using a machine learning algorithm by recasting the pointer to the array as a data (e.g. integer) type or by using the original data pointer. This process is iteratively repeated until an end criterion is reached.

The entities evolve in such a manner as to improve their fitness, and one entity is ultimately produced which represents the best solution to the problem.

Each entity includes a plurality of directly executable machine code instructions, a header, a footer, and a return instruction. The alteration process is controlled such that only valid instructions are produced. The headers, footers and return instructions are protected from alteration.

The system can be implemented on an integrated circuit chip, with the entities stored in high speed memory or in special on chip registers in a central processing unit and with the creation initialization, storage, alteration and execution operators stored in chip based microcode or on ROM.

The present invention overcomes the drawbacks of the prior art by eliminating all compiling, interpreting or other steps that are required to convert a high level programming language instruction such as a LISP S-expression into machine code prior to execution. It overcomes the need of compiler based systems to access, repeatedly, the Learned Elements as run-time data, which greatly slows down the system. It also represents the problem to the computer in the native language of the computer, as opposed to high level languages that were designed for human,

14

not for machine, understanding. It acts directly on the native units of the computer, the CPU registers, rather than through the medium of interpreters or compilers. It is believed that by using an intermediate process such as interpretation or compilation of high level codes, previous systems may actually interfere with the ability of computers to evolve good solutions to problems.

Experiments have demonstrated that the present approach can speed up the execution of a machine learning system by 1,000 times or more as compared to systems which provide potential solutions in the form of high symbolic level program expressions and by 60 times or more as compared to compiler based systems. This speed advantage makes possible practical solutions to problems which could not heretofore be solved due to excessive computation time. For example, a solution to a difficult problem can be produced by the present system in hours, whereas a comparable solution might take years using conventional techniques.

These and other features and advantages of the present invention will be apparent to those skilled in the art from the following detailed description, taken together with the accompanying drawings, in which like reference numerals refer to like parts.

Detailed Description

FIG. 1 illustrates a computer system 10 for implementing a machine learning method according to the present invention. The system 10 includes a Random Access Memory (RAM) 12 in which the software program for implementing the functionality of the invention is stored, and a processor 14 for executing the program code. The system 10 further includes a visual display unit or monitor 16 for providing a visual display of relevant information, a read-only memory (ROM) 18 for storing firmware, and an input-output (I/O) unit 10 for connection to a printer, modem, etc.

The system 10 further includes a mass data storage 20 which

15

can be any combination of suitable elements such as a fixed (hard) magnetic disk drive, a removable (floppy) disk drive, an optical (CD-ROM) drive, etc. Especially large programs which implement the invention may be stored in the storage 20, and blocks of the programs loaded into the RAM 12 for execution as required.

User access to the system 10 is provided using an input device 22 such as alphanumeric keyboard 114 and/or a pointing device such as a mouse (not shown). The elements of the system 10 are interconnected by a bus system 24.

The system 10 is preferably based on the SUN SPARC architecture due to its register structure. The invention is also preferably implemented using the "C" programming language due to its freedom in the use of data types and the ability to cast between different types, especially pointers to arrays and pointers to functions and a standard SUN operating system compiler, although the invention is not limited to any particular configuration.

The invention has been practiced using the above configuration in the SPARC environment which generally a stable architecture for low level manipulation and program patching. The particular platform used was the SUN SPARCSTATION 1+.

The RAM 12 is illustrated in FIG. 2 as storing an operating system 30 such as UNIX or one of its variants, and a program 32 written in the "C" language which provides the functionality of the invention. The program 32 includes high level machine language instructions 32a, and one or more machine code array 32b.

The high level instructions 32a are not directly executable by the processor 14, but must be converted into directly executable machine code by a compiler. The array 32b, however, is provided in native (lowest level) machine code including binary instructions that are directly recognized and executed by the processor 14.

The performance of the present invention is greatly enhanced by implementation thereof using a processor having

16

banks of registers. This architecture is provided by the SPARC system as illustrated in FIG. 3.

The components of the SPARC processor 14 which are most relevant to understanding the present invention include an Arithmetic Logic Unit (ALU) 40 for performing actual numeric and logical computations, a program counter 42 for storing an address of a next instruction to be executed, and a control logic 44 which controls and coordinates the operations of the processor 14 including memory access operations.

The processor 14 further includes banks of registers which are collectively designated as 46. The processor 14 also includes a number of additional registers which are not necessary for understanding the principles of the present invention, and will not be described. Although the invention is presently implemented using integer arithmetic, it may be applied to Floating Point arithmetic easily and such an application is within the scope and spirit of the invention.

More specifically, the registers 46 include eight register banks (only one is shown in the drawing), each of which has 8 output registers O_0 to O_7, 8 input registers I_0 to I_7, 8 local registers L_0 to L_7. There are, also, 8 global registers G_0 to G_7 accessible from any bank. The implementation of the invention using this architecture will be described below.

FIG. 4 is a flowchart illustrating the general machine learning method of the present invention which is implemented by the high level instructions 32a in the C programming language. Figures 22a to 22k are more detailed flowcharts illustrating the present invention. The first step is to define a problem to be solved, a fitness criterion for evaluating the quality of individual solutions, and an end criterion. The end criterion can be a predetermined number of iterations for performing the process (e.g. 1,000), achievement of a predetermined fitness level by one or more solutions, a change in total or individual fitness which is smaller than a predetermined value, or any other suitable criterion.

Next, the input, output, calculation, and state registers must be identified.

An input register is a register that is initialized before each calculation with data. In the present implementation of the invention, the input registers may be any one of the registers referred to as I0 through I5 in the Sun architecture. See Figure 3, number 46. In the present implementation of the invention, it is, therefore possible to have up to six input registers. However, it would be easy to extend the system to have many more input registers and such an extension is within the scope and spirit of the present invention. In any event, the designer must pick how many of the eight input registers to use.

There are many ways to initialize these registers but the preferred implementation is to include the input variables as parameters when the individual is called as a Native Function. For example, if native_function_array is an array of integers that constitute valid machine code instructions that amount to a Native Function and that amount to an entity in this invention, then the following type definition and function call execute the native_function_array individual and pass two input variables to it.

```
Typedef unsigned int(* function-ptr) ();
{the type definition}
a=((function_ptr) native_function_array)(2,3);
{calls the function and passes two inputs (2 and 3), which
are placed in I0 and I1.}
```

The items in curly brackets are descriptive and not part of the code. In the present implementation of this invention, the two input variables are put into I0 and I1. Accordingly, if the designer decides to have two input registers, they must be I0 and I1 in the present implementation. Other methods of initializing registers such as putting instructions in the header of the entity (see Figure 5, number 50) that retrieve data from RAM provide more flexibility as to what registers are input registers and more input registers and they are within the scope and spirit of the present invention.

This process of picking and initializing input registers is referred to in Figures 16b, c, and d; 17b, c, and d; 18b, d, and f; and 22b, e, and g.

An output register is read after the individual has finished executing on a given set of inputs. In other words, an output register is the output of the individual on a given set of inputs. The designer must designate one or more registers to be the output register(s) and the number depends on the problem to be solved. In the present implementation, the value in I0 at the end of the execution of the individual is automatically set as the single output register and as the output of the individual. The above code automatically puts the value in I0 into the variable "a." It is possible to designate multiple output registers and to preserve the values therein when an individual has been executed. One such method would be to include instructions in the footer of the individual (see Figure 5, number 50) that move the value(s) in the designated output registers in the individual that has just executed to storage registers or to locations in RAM memory. Such implementations are within the scope and spirit of the present invention.

This process of picking and recovering the values in output registers is referred to in Figures 16b, c, and d; 17b, c, and d; 18b, d, and f; and 22b, e, and g.

A calculation register is a register that is neither an input nor an output register but that is initialized to a given value every time the individual is executed. Zero or one is the preferred initialization value. A calculation register may be initialized in the headers of the individuals. (See Figure 5, number 50). In any event the designer must decide how many, if any, calculation registers he or she wishes to use and then designate specific registers to be calculation registers. So, for example, if there were two input registers, I0 and I1 and if I0 were the output register, then I2 through I5, and L0 through L7, among others would be available slots for a calculation

19

register. So if the designer desires one calculation register, he could pick I2 as the calculation register.

This process of picking and initializing calculation registers is referred to in Figures 16b, c, and d; 17b, c, and d; 18b, d, and f; and 22b, e and g.

A state register is a register that is neither an input nor an output register but that is not initialized every time the individual is executed. That is, values from some previous execution of the individual are retained in a state register. The precise retention policy is a matter of decision for the designer. At the beginning of the execution of the individual, the state register has to be initialized to the value it held at the end of a previous execution of the individual. If that value was saved in a storage register or in RAM, then the header of the individual (See Figure 5, number 50) must have an instruction moving the value from the storage register or RAM to the state register or the previous state could be passed to the individual as a parameter when the individual is called as a Native Function. In this case, the footer of the individual may have an instruction saving the value of the state register at the end of execution to the storage register or to RAM. There are many available ways to implement state registers and to save and initialize their values and all of them are within the scope and spirit of this invention. In any event the designer must decide how many, if any, state registers he or she wishes to use and then designate specific registers to be state registers. So, for example, if there were two input registers, I0 and I1, and one calculation register, I2, and if I0 were the output register, then I3 through I5, and L0 through L7, among others would be available for a state register. So if the designer desires one state register, he or she could pick I3 as the state register.

This process of picking and initializing state registers is referred to in Figures 16b, c, and d; 17b, c, and d; 18b, d, and f; and 22b, e and g.

Once all registers have been designated, the designer has

20

established the "Register Set." In the example above, the Register Set would be as follows

T210x

input	I0, I1
output	I0
calculation	I2
state	I3

Once the Register Set has been picked, it is essential that all individuals be initialized with instructions that contain only references to valid members of the register set. See Figures 16c, 17c, 18d, and 22e. It is also essential that all changes to the individuals, including those made with genetic operators, modify the individual in such a way as to make sure that all register references in the modified instructions are to valid members of the Register Set.

Next, a solution or a population of solutions is defined as an array of machine code entities. These entities are preferably "C" language functions which each include at least one directly executable (machine language) instruction. An instruction is directly executable by being constituted by the lowest level (native) binary numbers that are directly recognized by the control logic 44 and ALU 40 as valid instructions. These instructions do not require any compilation, interpretation, etc. to be executable.

The invention will be described further using the exemplary case in which the entities are "C" language functions. However, it will be understood that the invention is not so limited, and that the entities can be constituted by any data structures including machine code instructions that can alternatively be manipulated as data and executed by a higher level program.

After the function or array of functions has been defined, they are initialized by inserting valid machine code instructions in the appropriate locations therein. As illustrated in FIG. 5, a function 50 includes a header 50a, an instruction body 50b, a footer 50c, and a return instruction

50d.

The header 50a deals with administration which is required when entering a function. This normally involves manipulation of the registers 46, including passing arguments for the function to and from the various registers 46, saving the values in the registers that exist when the function is called and saving the address of the calling instruction so that, when the function is finished, program execution can begin from where it left off. The header can also be used to perform many other useful functions at the option of the designer including initializing registers and recovering the values of saved state registers. There can also be processing to ensure consistency of processor registers.

The header 50a is often constant and can be added at the beginning of the initialization of the individual machine code functions in the population. Mutation, crossover and any other operator that alters the entity must be prevented from changing this header field when they are applied to an individual function or the field must be repaired after the application of the operator.

The footer 50c is similar to the header 50a, but does the operations in the opposite order and "cleans up" after the function call by, among other things, restoring the registers to their state before the function call, recovering the output of the function, saving state registers, and many other useful functions at the option of the designer. The footer field must also be protected from change by the genetic or other alteration operators or the field must be repaired after the application of the operator.

The return instruction 50d forces the system to leave the function and return program control to the calling procedure via the program counter 42. If variable length programs are desired, the return instruction can be allowed to move within a range defined by a minimum and maximum program size. The return instruction must also be protected from change by the genetic or other alteration operators or the field must be repaired after the application of the operator.

The function body 50b includes at least one directly executable machine code instruction, for example instructions 52a, 52b and 52c. In the SPARC architecture, instructions have a fixed length of 32 bits. Instructions can be of two types: a first type which is designated as 54 and consists of a single 32 bit operator; and a second type which is designated as 56 and includes a 19 bit operator and a 13 bit operand. The operand represents data in the form of a numerical variable.

During the initialization step, the instruction bodies 50b of the functions 50 are filled with valid machine code instructions in a suitable manner. In the preferred implementation, the instructions are initialized from a Default Instruction Template Set, which contains partially "blank" instructions that contain information on what types of instruction is represented by a particular template. For example, is it an instruction that adds the values in two registers together or an instruction that adds a constant to a register value? What is left blank in the template is the register(s) to which the instruction applies and the constant values to be included in the instruction. The register references and the constant values may be added by methods such as those described in Figure 18d. In Figure 18d, instruction templates are randomly chosen and then the blanks are filled out so that the integer array constituting the individual contains only valid machine code instructions, taking care to limit the register references therein members of the Register Set. See Figures 17c, 18b and 18d. The instructions can be selected randomly, or using any suitable selection criterion.

In the most basic form of the invention, to limit the search space and to avoid complex control mechanisms and thus achieve maximum performance, the set of machine code instructions is limited in a number of ways. Only two registers and only those machine code instructions of two addressing mode types are used. All of them operate internally in the processor. However, more lengthy and complex implementations of

23

the Register Set are described above and more complex implementations of the instruction set are within the spirit and scope of the invention.

The first addressing mode takes one argument from memory immediately afterwards and performs an operation on this argument and a processor register and leaves the result in a register. The other takes the operand from the second register, performs the operation and leaves the result in the first register.

With these kinds of instructions it is possible to reach the maximum processor throughput. The instructions are also constrained to those which take an integer as argument and return an integer. This however does not limit the problem space to mathematical problems or just to integer arithmetic. No control instructions such as jump instructions are allowed which means that no loops can be formed. These limitations reduce the complexity and thus execution time of the individual programs.

The basic instructions are listed in the following TABLE.

T2-2x

ADD (imm), ADD (reg2)	Add register one to an immediate operand or add register one to register two.
SUB (imm), SUB (reg2)	Subtract register one from an immediate operand or subtract register one from register two.
MUL (imm), MUL (reg2)	Multiply register one by an immediate operand or multiply register one by register two.
DIV (imm), DIV (reg2)	Divide register one by an immediate operand or divide register one by register two.

24

OR (imm), OR (reg2)	Logical OR of register one and an immediate operand or logical OR of register one and register 2.
AND (imm), AND (reg2)	Logical AND of register one and an immediate operand or logical AND of register one and register two.
XOR (imm), XOR (reg2)	Logical EXCLUSIVE OR of register one and an immediate operand or logical EXCLUSIVE OR of register one and register two.
SETHI	Set the high bits of a register, used when an operand bigger than 13 bits hits needs to be loaded into a register.
SRL	Logical shift right of register one to an immediate operand or logical shift right of register one a number of steps defined by register two.
SLL	Logical shift left of register one to an immediate operand or logical shift left of register one and register two.
XNOR	Logical EXCLUSIVE NOR of register one to an immediate operand or logical EXCLUSIVE NOR of register one and register two.

TABLE

These functions can and are used to implement other common processor functions. Loading of operands into registers is performed by the OR instruction, clearing a register is performed by the AND instruction, and the negation instruction can be performed by the XNOR instruction etc.

2

An exemplary array 60 of functions F_0 to F_6 is illustrated in FIG. 7. Each function includes a header 50a including instructions which are collectively designated as H, a body 50b including instructions B, a footer 50c including one or more instructions F, and a return instruction 50d which is indicated as R. Although the figure shows a length of three instructions for the header and one for the footer, those particular numbers are for illustrative purposes only.

The functions of the array 60 are illustrated as having a fixed length. However, the invention is not so limited, and the functions can have variable lengths.

FIG. 8 illustrates an alternative embodiment of the invention in which only a single solution in the form of a function 62 is provided. The function 62 is configured as a continuous array that contains valid machine code instructions, and is typically larger than the functions of the array 60. Such a single array would be a typical way to implement a system not involving machine learning but involving repeated computations on run time data. Certain machine learning approaches would use such a single structure also.

After the array is initialized, it is recast as a function type array, and the registers 46 are initialized with training data. The training data is normally part of a "training set", each element of which consists of one or more inputs and a desired output. The training set represents the problem that is to be solved. However, the invention is not so limited, and the training data can alternatively include testing, validation, prediction, or any other data suitable for machine learning or for repetitive calculations on run-time data.

For the machine learning applications, the purpose of the learning process is to train the functions using the training data by causing the functions to evolve until one of them produces outputs in the output register(s) in response to the training inputs in the input register(s) that closely approximate the desired outputs. The inputs are passed to the functions by initializing the registers with the training data as described elsewhere.

The functions of the array (or single function as illustrated in FIG. 8) are then executed with the training data as inputs. This execution causes the functions to produce outputs which are calculated in accordance with the instructions in the functions.

The output register or registers are then read, stored and compared with actual or desired outputs from the training data to calculate fitnesses of the functions. A function has a high fitness if its output closely approximates the desired output, and vice-versa. If the end criterion is met, the program ends.

If the end criterion is not met, the array is recast or used as a data array (e.g. integer), and the instruction bodies 50b are altered. During this process, selected individual instructions are altered, replaced, added or deleted such that the altered function includes at least one valid machine code instruction. In other words, regardless of the manner in which a particular function is altered, the program ensures that the altered function will not include any invalid instructions.

The function(s) can be altered using any machine learning algorithm. One preferred methodology is evolution of machine code structures and machine code contents using crossover and mutation operators as will be described in detail below. However, the invention is not limited to any particular learning paradigm, nor is it limited to machine learning applications. The invention can be used, for example, to alter run-time data encoded in the machine code for the purpose of performing repetitive calculations on the run-time data.

As described above, it is important that the header, footer and return instruction will not be altered. This can be accomplished by protecting these elements from alteration using masking or other programming techniques. Alternatively, these elements can be allowed to be altered, and their initial states subsequently restored by repair or replacement. As another alternative, the instructions can be separated from the function, altered, and then returned to the function. There are

27

other ways to protect the header, footer and return instructions from being altered, all of which are within the scope and spirit of this invention.

After performing the alteration, the program loops back to the step of recasting the array as a function and executing the function. The program iteratively repeats these steps until the end criterion is met.

The output of the system, when used for machine learning, is the function that has the highest fitness and thereby produces outputs that are closest to the desired outputs in response to the training inputs. The functions evolve as the result of the alterations, with one individual finally emerging as being improved to the highest fitness. The policy for selecting a "best individual(s)" is a matter of discretion by the designer. This system applies to any such policy.

As described above with reference to FIG. 3, the registers 46 include banks of input registers and output registers. These registers are used to pass data from a calling function to a called function, and are used to initialize the functions with training data as further described above.

More specifically, functions use their input registers to store variables. When a function calls another function, the Spare Architecture transfers the contents of its input registers to its output registers. The called function transfers the contents of the calling function's output registers into its own input registers, and operates on the data using its input registers. The opposite operation takes place when control is returned from the called function to the calling function.

Thus, the C program initializes a function with training data by storing the training data in its output registers and then calling the function. This method of initialization is preferred in the SPARC implementation, but the invention is not so limited. Other mechanisms for initializing functions with training data may be preferable with other computer architectures or for more complex application on the SPARC architecture.

An important feature of the present invention is to create and alter machine code functions as data, and execute the functions, from a high level language such as C. The following illustrates how this manipulation can be accomplished using the SPARC architecture.

As a simplified example, define a simple function that computes the sum of two input parameters and returns the result from this addition.

sum(a,b) = a+b

This function can be translated by a compiler in the SPARC architecture to the following five assembly language instructions.

```
save
add \%i0,\%i1,\%i0
restore
ret
nop
```

These five assembly language instructions are stored sequentially in memory as five 32-bit numbers having the following decimal equivalents.

```
2178940928
2953183257
2179465216
2177359880
16777216
```

The "save" instruction corresponds to the number 2178940928, and is a special machine code instruction in the SPARC architecture that saves the contents of the calling function's registers and transfers the parameters to the called function.

The "add" instruction adds the contents of input register "i0" to the contents of input register "i1", and stores the result in input register "i0". This instruction is coded as the integer 2953183257. Additions between other registers are represented by other integers.

The "restore" instruction restores the registers from the calling function and transfers the result from the addition (present in register i0) for access from the calling function.

The return instruction "ret" jumps back to the memory location from where the function was called.

The "nop" instruction is a no operation instruction that does nothing but "entertain" the processor while jumping back. This instruction could be left out if the order of the "restore" and "ret" instructions was reversed as will be described below.

Implementation of the present invention involves calling a machine code function as exemplified above from the C or other high level language. Adding the numbers 2 and 3 by calling the function sum(2,3) can be represented in assembly language as follows.

```
...
mov 2,\%o1
mov 3,\%o0
call _sum
nop
...
```

These instructions are represented by these four integers:

```
2450530306
2484084739
2147483646
16777216
```

The first instruction stores the input parameter "2" in the output register %o1. The second instruction stores the input parameter "3" in the output register %o0.

The call instruction jumps to the location of the sum function and stores the address of itself in a register (output register %o7 by convention). This causes the program counter 42 to contain the address or location of the first instruction of the sum function. In other words, it contains the address of a memory location which stores the integer 2178940928 (save), and execution continues from here until the return instruction is encountered. After the return from the function, the output register %o0 will contain the result from the summation which can be used in further calculations.

When a save instruction is executed, it copies the contents of all output registers to an unused set of input registers. In this way the sets of input and output registers are used as a small internal stack for transfer of parameters and saving of register contents. The calling function stores the parameters of the called function in the output registers, the save instruction copies them to the input registers of the called function, and the restore instruction later copies all input registers of the called function into the output registers of the calling function. If the number of parameters is larger than the number of registers, the memory in the stack has to be used, as in most other processors.

In this manner, the input data (training data) is passed to the array of functions (solutions) by storing the data in the output registers of the calling function, and then calling the array.

The "nop" does nothing but keep the processor occupied while jumping to the function. In a more effective but less clear version, the "mov 3,%o0" instruction is placed after the "call", which makes the processor execute this instruction "while" jumping to the function.

The following illustrates how to call the sum function defined above from the C-language.

```
typedef unsigned int(* function_ptr) ()
unsigned int sumarray[]={2178940928, 2953183257, 2179465216,
2177359880, 16777216};
a=((function_ptr) sumarray)(2,3);
```

These three lines of C-code will compute the sum of 2 and 3 and put the result in a variable "a" by calling a function defined by the integers in the "sumarray".

The first line of code defines the function pointer type in C, because this type is not predefined in the C-language.

The second line of code declares an integer array containing integers for the instructions in the sum function as defined above.

31

The last line of code converts the address of the sumarray from a pointer to an integer array to a pointer to a function, and then calls this function with the arguments 2 and 3. The result from the function call is placed in variable "a".

This example illustrates the basic principles of manipulating binary programs. The sumarray can be altered like any other array as long as its contents are valid functions that can be called (executed) to evaluate the results of the algorithm.

The present method as implemented in the C programming language utilizes four instructions for initializing, altering, and executing machine code functions.

1. A first instruction that points to and designates machine code stored in a memory as data.

2. A second instruction that points to and designates machine code stored in the memory as at least one directly executable function.

3. A third instruction that alters machine code pointed to by the first instruction.

4. A fourth instruction that executes machine code pointed to by the second instruction.

Examples of the four instructions are presented below.

1. unsigned int * theintegerpointer;

{declaration of integer pointer variable}

theintegerpointer = (unsigned int *) malloc(Max_Individual_Size * Instruction_Size)

{The instruction creating a pointer to an integer array}

2. Alternative 1

----

typedef unsigned int(*function_ptr) ();

{definition of the function_ptr type as a pointer to a function}

function_ptr thefunctionpointer

{declares thefunctionpointer as a variable of type function_ptr)

[illegible]

thefunctionpointer=(function_ptr) theintegerpointer;
{Instruction that typecasts theintegerpointer as a pointer to a function}

Alternative 2

----

typedef unsigned int(*function_ptr) ();
{definition of the function_ptr type as a pointer to a function}

Predicted_Output = ((function_ptr) theintegerpointer)(Input Data Here);
{the code "((function_ptr) theintegerpointer)" is an expression within an instruction that performs a typecast.}

3. theintegerpointer[2] = theintegerpointer[2] ¦ 1234;
{XOR's the value in the second instruction with the integer 1234}

or

theintegerpointer[3] = 16777216;
{places an nop instruction in the third instruction slot, overwriting the previous instruction}

4. var_a = thefunctionpointer(Input Data Here);

or

Predicted_Output= ((function_ptr) theintegerpointer)(Input Data Here);

In the CGPS embodiment of the invention which utilizes genetic crossover and mutation operations, the types of operations that can be performed depend on the type of instruction; more specifically if the instruction includes only an operator, or if it includes an operator and an operand. Examples of genetic crossover and mutation operations are illustrated in FIGs. 10 to 15.

FIGs. 10a and 10b illustrates a uniform crossover operation in which like numbers of adjacent complete instructions are exchanged or swapped between two functions. FIG. 10a illustrates two functions 70 and 72. Uniform crossover is performed by exchanging, for example, two instructions indicated as "4" and "2" in the function 70 for two instructions indicated as "5" and "9" in the function 72.

The results are illustrated in FIG. 10b. An altered function 70′ includes all of its original instructions except for the "4" and the "2" which are replaced by the "5" and the "9" from the function 72. The function 72 includes all of its original instructions except for the "5" and the "9" which are replaced by the "4" and the "2" from the function 70.

FIGs. 11a and 11b illustrate "2-point crossover", in which blocks of different numbers of complete instructions are exchanged between two functions. In this case, two points are selected in each function, and all of the instructions between the two points in one function are exchanged for all of the instructions between the two points in the other function.

In the example of FIG. 11a, instructions indicated as "7" and "8" in a function 74 are exchanged for instructions "4", "7", "6", and "1" in a function 76 to produce functions 74′ and 76′ as illustrated in FIG. 11b.

FIGs. 12a and 12b illustrate how components of functions can be crossed over. In FIG. 12a, two instructions 78 and 80 have operators 78a and 80a and operands 78b and 80b respectively. In this case, the operator 78a of the function 78 which is indicated as "OP1" is exchanged for the operator 80a of the function which is indicated as OP2. The result of the crossover is illustrated in FIG. 12b.

FIGs. 13a and 13b illustrate an example of how uniform crossover can be performed between all or parts of operands. In FIG. 13a, a function 82 has an operator 82a and an operand 82b, whereas a function 84 has an operator 84a and an operand 84b. The rightmost two bits of the operand 82b are exchanged for the rightmost two bits of the operand 84b to produce functions 82′ and 84′ with operands 82′ and 84′ as illustrated in FIG. 13b.

FIG. 14 illustrates how the operator of a function 86 can be mutated. In this case, the function 86 initially has an operator which is indicated as OP1, and is altered or replaced so that a mutated function 86′ has an operator OP2. It is necessary that both operators OP1 and OP2 be valid machine code instructions in the set of instructions used by the system.

FIG. 15 illustrates how all or part of an operand can be mutated. A function 88 has an operator 88a and an operand 88b. In this example, the second least significant bit of the operand 88 is changed from "1" to "0" or "flipped" to produce an altered function 88′ having an altered operand 88b′.

As set forth above, the present invention can be applied to any applicable problem by using any suitable machine learning algorithm or to any problem involving repeated calculations on run-time data. The principles described in detail above can be applied to implement a particular application and computing environment.

In addition to implementing the present machine learning system on a general purpose computer as described above, it is possible to implement part or all of the system as a specially designed integrated circuit chip 90 such as an Application Specific Integrated Circuit (ASIC) which is symbolically illustrated in FIG. 9.

The chip 90 comprises a Central Processing Unit (CPU) 92 including a processor 94 and a RAM 96. The processor 94 includes normal CPU microcode plus microcode implementing storage, initialization, creation, and alteration operators. The RAM 96 is preferably a high speed cache memory which the processor 94 can access at a much higher speed than the processor 94 can access off-chip memory.

The RAM 96 can include a number of registers, or can have a conventional address based architecture. Preferably, the population of functions (solutions) is stored in the RAM 96 for rapid manipulation and execution by the processor 94. Other alternatives include additionally storing the high level program in the RAM 96. As yet another alternative, the chip 90 can include a ROM 98 for storing, for example, a kernel of the high level program.

The on chip memory, alternatively could be registers dedicated to storing individuals or high speed, on chip RAM that is not a cache. In addition, the CPU could alternatively execute machine learning operators such as crossover and mutation or any other operators that initialize, create, evaluate or alter individuals in microcode or in high speed ROM.

A preferred implementation of the invention evolves machine code structures and machine code contents as a way of learning the solution to a problem. A detailed flowchart of this system is presented in FIG. 6 as an example for reference purposes.

The program utilizes a small tournament in combination with genetic crossover and mutation operations, and includes the following basic steps.

1. Randomly pick four individuals from the population.

2. Evaluate them in pairs, two at a time according to their fitness.

3. Let the two winners breed.

4. Replace the losers with the children of the two winners.

5. Repeat step 1-4 until the success predicate is true or the maximum number of tries is reached.

The flowchart is believed to be self-explanatory except for probabilistically choosing one operation or another. These choices are analogous to flipping a coin, and can be implemented using, for example, a random number generator which generates random numbers between 0 and 1. If the number is 0.5 or less, a first probibilistic branch Pa is taken. If the number is higher than 0.5, a second probibilistic branch Pb is taken. Choices having more than two options can be implemented by dividing the range between 0 and 1 into a number of subranges equal to the number of choices.

FIGs. 16a to 16d are further provided for reference, and in combination constitute a detailed flowchart of the generic machine learning system of the present invention. The entries in the flowchart are believed to be self-explanatory, and will not be described in detail.

FIG. 16a outlines the general operation of the system. FIG. 16b sets forth the details of the block entitled "SYSTEM DEFINITION" in FIG. 16a. FIG. 16c sets forth the details of the

[illegible]

block entitled "INITIALIZATION" in FIG. 16a, whereas FIG. 16d sets forth the details of the block entitled "LEARN FOR ONE ITERATION in FIG. 16a.

Figures 17a through 17d are further provided for reference, and in combination constitute a detailed flowchart of the application of the invention to any computation problem that involves repeated access to run-time data. The entries in the flowchart use the terminology of this application and are believed to be self explanatory in the context of this application, and will not be described in detail.

Figures 18a through 18g are further provided for reference, and in combination constitute a detailed flowchart of the application of the invention as a system that induces the solution to problems by learning the structure and content of machine code entities. The entries in the flowchart use the terminology of this application and are believed to be self explanatory in the context of this application, and will not be described in detail. The mutation and crossover operators referred to in said Figure 18g are the operators described elsewhere in this application.

FIG. 16a outlines the general operation of the system when it is applied to a generic machine learning problem.

FIG. 16b sets forth the details of the block entitled "SYSTEM DEFINITION" in FIG. 16a. The steps in this figure show, inter alia, what steps to take to analyze any machine learning problem to permit the designer to encode entities that contain both the Learned Elements and the Conversion Elements into a machine code entity that can be created, initialized, stored, modified and executed by the means described in this application. Numeric or other values may be encoded in the machine code as constants.

FIG. 16c sets forth the details of the block entitled "INITIALIZATION" in FIG. 16a. This Figure sets forth, inter alia, a set of steps that will result in the creation of one or more learning entity or entities that will be the learning entity or entities in the machine learning system that is being implemented using the present invention. Such entity or entities will be created, stored, initialized, modified and executed by the methods set forth herein but when and how such steps take place will be set according to the requirements of the particular machine learning algorithm being implemented, as shown in FIG. 16d. It is important to note that the entity created according to the procedure outlined in Figure 16c ordinarily includes not only the Conversion Elements but also contains the first set of Learning Elements for evaluation. Should said first set not be available when the initialization occurs, then the entity should be initialized with dummy values in the places in the instructions where the real Learning Elements will reside. Then, when the first set of real Learning Elements are known, it can be placed into those places using the procedure under "Start Modification" in FIG. 16d.

FIG. 16d sets forth the details of the block entitled "LEARN FOR ONE ITERATION" in FIG. 16a. This figure shows, inter alia, how to modify and how to evaluate an entity when the particular machine learning algorithm being implemented calls for either of those steps. The particular implementation of an application of the invention to a particular machine learning problem will vary substantially from problem to problem and among various machine learning systems. So this Figure is general in terms of when to evaluate the entity (referred to in FIGs. 16a-16d as a "solution") and when to modify the entity. Because of the breadth of various approaches to machine learning, this Figure indicates that steps other than the ones shown specifically are appropriate and that systems including such other steps are within the spirit and scope of the present invention.

FIG. 17a outlines the general operation of the invention when the invention is to be used to perform repeated computations on run-time data. This application of the invention could be handled in other and alternative manners both in the general matters set forth in FIG. 17a and in 17b through 17d and those other and alternative manners are within the scope and spirit of this invention.

FIG. 17b sets forth the details of the block entitled "SYSTEM DEFINITION" in FIG. 17a. This Figure shows, inter alia, how to define a repeated computation on run-time data problem so that it may be implemented using the invention. Numeric or other values may be encoded in the machine code as constants.

FIG. 17c sets forth the details of the block entitled "INITIALIZATION" in FIG. 17a. This Figure sets forth, inter alia, a set of steps that will result in the creation of a computational entity that will perform repeated computations on run-time data based on the system definition performed in FIG. 17b. It is important to note that the computation entity created ordinarily includes not only the related code but also contains the first set of run-time data on which the system will perform repeated calculations. Should said first set not be available when the initialization occurs, then the computational entity should be initialized with dummy values in the places in the instructions where the real run-time data will reside. Then, when the real run-time data is known, it can be placed into those places using the procedure under "Start Modification" in FIG. 17d.

FIG. 17d sets forth the details of the block entitled "EVALUATE OR MODIFY FUNCTION" in FIG. 17a. The particular implementation of an application of the invention to repeated computations on run-time data will vary substantially from problem to problem. So this Figure is general in terms of when to execute the computational entity (referred to in FIGs. 17a-17d as a "solution") and when to modify the computational entity. The computational entity should be modified each time that the designer wishes to perform a particular computation (related code) on run-time data of the type that is encoded in the computational entity but that has not yet been placed in the computational entity. The computational entity should be executed every time the designer wants to perform a calculation on run-time data that has been encoded into the machine code of the computational entity.

39

FIG. 18a outlines the general operation of the invention when it is applied to learning machine code structures and machine code contents that will operate a register machine for the purpose of learning a solution to a problem. This application of the invention could be handled in other and alternative manners both in the general matters set forth in FIG. 18a and in 18b through 18h and those other and alternative manners are within the scope and spirit of this invention.

FIG. 18b sets forth the details of the block entitled "SETUP" in FIG. 18a. This Figure describes various steps necessary to setup an application of the invention to a CGPS run on a particular problem. Numeric or other values may be encoded in the machine code as constants.

FIG. 18c sets forth the details of the block entitled "INITIALIZATION" in FIG. 18a. This Figure describes, inter alia, a method for initializing a population of entities for the purpose of conducting a CGPS run.

FIG. 18d sets forth the details of the block entitled "CREATE INSTRUCTION" in FIG. 18c. This Figure describes, inter alia, one method of creating a single machine code instruction to be included in the body of an entity.

FIG. 18e sets forth the details of the block entitled "MAIN CGPS LOOP" in FIG.18a. This Figure sets forth, inter alia, one approach to implementing the basic CGPS learning algorithm.

FIG. 18f sets forth the details of the block entitled "CALCULATE INDIV[N] FITNESS**" in FIG.18e. The fitness function described therein is simple. More complex fitness functions may easily be implemented and are within the spirit and scope of this invention.

FIG. 18g sets forth the details of the block entitled "PERFORM GENETIC OPERATIONS . . ." in FIG. 18e. The mutation and crossover operators referred to in Figure 18g may be all or any of the operators described elsewhere in this application.

FIG. 18h sets forth the details of the block entitled "DECOMPILE CHOSEN SOLUTION" in FIG. 18a. This Figure describes a method of converting an individual into a standard C language

40

function that may then be linked and compiled into any C application. Other and more complex methods may be easily implemented and are within the spirit and scope of this invention.

TURING COMPLETE MACHINE LEARNING SYSTEM

A basic machine learning system and method according to the present invention is described above. The system can be made Turing complete though the addition of branching instructions as will be described in detail below. This capability increases Ins D4 the flexibility and power of the system by enabling subroutines, leaf functions, external function calls, recursion, loops, and other types of conditional operations.

Introduction

The modern computer has grown in instruction capabilities and machine word size, which means that there are more tasks that can be carried out elegantly by a straightforward machine language subroutines. For example, the precision of the arithmetic is enough for direct usage, and the memory organization is often flat which makes pointer operations less complex, etc.

Conventional application programs often fail to use the computer to its full potential because the algorithms are not implemented in the most efficient way. This fact is even more relevant in the use of algorithms for machine learning, artificial intelligence, artificial life, etc., repeated computations on run-time data which rely on complex manipulations and are often of the meta type where programs manipulate program structures.

The present machine learning system is a unique approach to implementation of algorithms of this kind. The approach is unknown, in spite of the fact that it is possible to use on the oldest and smallest computer. It can present a number of advantages like a speed up of a 1,000 times, low memory consumption, compact representation, etc.

The execution speed enhancement is important both for real-life applications, and for simulations and experiments in science. A large efficiency improvement of this magnitude can make real life applications feasible, and it could also mean the difference of whether an experiment or simulation will take three days or one year. This can make a given algorithm useful in a whole new domain of problems.

If an algorithm uses a representation in a program that later can be evaluated by some kind of interpretation as a program, even in a limited sense, this approach can be used. It could be the rules in Cellular Automata, the creatures in artificial life, decision trees, rules, simulations of adaptive behavior, or as demonstrated below evolving Turing complete algorithms with an evolutionary algorithm. As described above, the present approach is referred to as a compiling approach because there are no interpreting parts and the individuals are, in effect, directly compiled.

The present method of binary machine code manipulation should not be confused with translating a machine learning algorithm into assembly code, the way a compiler operates. Instead, the present invention is capable of meta manipulating machine code in an efficient way.

The Present Compiling Approach

It is surprising that there are no existing programming languages with specialized features and tools for this programming paradigm.

The present invention provides advantages of the prior art including the following.

Higher execution speed

Compact kernel

Compact representation

Uncomplicated memory management

Low complexity algorithms

The efficiency improvement in binary meta manipulations comes from deletion of the interpreting steps. Instead of using a structure defined by the programmer to be run through an interpreter, a match between the data structures of the application and the binary machine code structures is sought. The algorithm then manipulates the binaries and instead of interpreting, the system executes the binary directly. The efficiency benefit comes mostly from deletion of the interpreting steps, but also from the simplicity of manipulating the linear integer array that constitutes the binary machine code.

The lack of an interpreting or compiling step makes the kernel of the algorithm compact, because no definition of the interpretation is needed. The machine code binary format is often compact in itself, which is important when working with applications that use a large set of structures, for example evolutionary algorithms, cellular automata, etc.

The inherent linear structure of a binary code program forces the design of a system that uses uncomplicated memory management. The decreased use of pointer and garbage collection imposes a straightforward handling of memory that is an advantage in for example real-time applications.

Some parts of the machine learning algorithm might be simplified by the restriction of the instructions to integer arithmetic. Handling of a sequence of integers is a task that a processor handles efficiently and compactly, which could simplify the structure of the algorithm.

The approach of binary manipulating machine learning algorithms builds on the idea behind a von Neumann machine, and it is thus applicable on most of today computers from the largest super computers to the small invisible systems in cars, cameras and washing machines.

Von Neumann Machines

A Von Neumann machine is a machine where the program of the computer resides in the same storage as the data used by that program. This machine is named after the famous Hungarian/ American mathematician Jon von Neumann, and almost all computers are today of the von Neumann type. The fact that the program could be considered as just another kind of data makes it possible to build programs that manipulate programs, and programs that manipulate themselves.

The memory in a machine of this type could be viewed as an indexed array of integers, and a program is thus also an array of integer numbers. Different machines use different maximal sizes of integers. A 32-bit processor is currently the most commonly commercially available type. This means that the memory of this machine could be viewed as an array of integers with a maximum size of 2^{32}-1, which is equal to 4294967295, and program in such a machine is nothing more than an array of numbers between zero and 4294967295.

A program that manipulates another program's binary instructions is just a program that manipulates an array of integers. The idea of regarding program and data as something different is however deeply rooted in our way of thinking. It is so deeply rooted that most designers of higher language programmed have made it impossible for the programmer to access and manipulate binary programs. It is also surprising that no languages are designed for this kind of task. There are a no higher level languages that directly support this paradigm with the appropriate tools and structures.

The C language is desirable for practicing the present invention because it makes it possible to manipulate the memory where the program is stored.

The Processor

The processor is a "black box" which does the "intelligent" work in a computer. The principles of different available processors are surprisingly similar. The processor consists of several parts as illustrated in FIG. 3, including the control logic 44 which access the memory 12, the ALU 40, and the registers 46.

The control logic 44 uses a register or some arithmetic combination of registers to get an index number or address. The content of the memory array element with this index number is then placed in one of the registers of the processor.

A register is a place inside the processor where an integer can be stored. Normally a register can store an integer with the same size as the so-called word size of the processor. A 32-bit processor have registers that can store integers between 0 and 4294967295.

The most important register is the program counter (PC) which stores the address of the next instruction to be executed by the processor. The processor looks at the contents of the memory array at the position of the program counter and interprets this integer as an instruction that might be an addition of two registers or placing a value from memory into a register.

An addition of a number to the program counter itself causes transfer of control to another part of the memory, in other words a jump to another part of the program. After doing an instruction the program counter is incremented by one and another instruction is read from memory and executed.

The ALU in the processor perform arithmetic and logic instructions between registers. All processors can do addition, subtraction, logical "and", logical "or", etc. More advanced processors do multiplication and division of integers, and some have floating point units with corresponding registers.

Every behavior we can see in modern computers is based on these simple principles. Computers doing graphics, animations, controlling a washing machine or watching the ignition system in a car all does these memory manipulations and register operations.

The principles above do not anywhere prevent a program from manipulating the contents of memory locations that later will be placed in the program counter and run by the processor as it interprets it as instructions. This is the basis of the binary manipulating machine learning code in accordance with the present invention.

<u>Machine Code and Assembly Code</u>

Machine language is the integers that constitute the program that the processor is executing. These numbers could be expressed with, for example, different radix such as decimal, octal, hexadecimal or binary. By binary machine code we mean the actual numbers stored (in binary format) in the computer.

When programming or discussing machine language, it is often impractical to use numbers for instructions. To remember that the addition instruction for example is represented by the integer 2416058368 in the SUN SPARC architecture is not natural to the human mind.

If we represent it in hexadecimal radix (E219922D), it will be more compact and it is more easy to deal with than its binary equivalent, but it is still not natural to remember. For this reason, assembly language has been developed. Assembly language uses mnemonics to represent machine code instruction. For example, addition is represented by the three letters "ADD".

The grammar for assembly language is very simple, and the translation or mapping from assembly language to machine code is simple and straightforward. Assembly language is not, however, machine language, and cannot be executed by the processor directly without the translation step.

RISC and CISC

The present invention has been implemented using both processors of the CISC type including the Intel 80486, and of the RISC type including the SUN-SPARC architecture. CISC stands for Complex Instruction Set Computer, and RISC stands for Reduced Instruction Set Computer. As indicated by the acronym, the RISC processor has fewer and less complex instructions than the CISC processor. This means that the RISC processor can be implemented differently in hardware and that it therefore will be faster.

There are advantages of both paradigms when using them for binary manipulating machine learning. The CISC has the advantages of many more types of instructions and addressing modes, and a subset can re readily found which makes a particular implementation straightforward.

The RISC processor on the other hand has the advantage that the structure or "grammar" of an instruction is very simple, for example the number of bits is constant. This sometimes makes the manipulation easier. It is easier to check for a valid instruction. A RISC is also often faster than a CISC.

Although the present invention can be practiced using CISC and RISC systems, it is somewhat easier to administrate on an RISC based architecture.

Another important factor is the operating system and hardware support of the development environment. A system like UNIX which offers separate processes and hardware implemented memory protection is definitely to recommend during development, because it will save time when bugs appear during the development phase. These systems do not have to be restarted after a serious bug like an illegal instruction. It should, however, be noted that once the system is debugged it will run as safely on any platform as any other conventional program.

Structure of Machine Code Function

A procedure and a function can be regarded as very similar concepts in machine code or even in C. A procedure is a function that does not return a value. When implementing a machine learning system with a binary manipulating technique, functions are the basic structures. For example, the individuals in compiling genetic algorithms are implemented as machine code functions. A function call has to perform three different subtasks:

Program control transfer and saving the start address
Saving processor registers
Transfer parameters

The most important instruction for functions and procedures is the call instruction. It is present in all kinds of processors, and works as a jump instruction that saves the memory location or address of where it was jumping from. This will allow a return instruction to return back to this memory

41

location when execution of the function is complete.

Most call instructions save the return address in a special memory segment called the stack, but some save it internally in a register in the processor. The SUN SPARC architecture saves the return address in a register, if there is enough room for it.

A call instruction is not sufficient to make a complete function call. The contents of the registers must be saved somewhere before the actual instructions of the function are executed.

This assures that the called function will not interfere with the processing in the calling function, and gives the called function the liberty to manipulate these registers itself. The most common place to store the contents of the calling functions registers, is the stack. Some architectures like the SPARC architecture stores it inside the processor by the special save instruction.

When the execution of the function is done, the registers of the calling function have to be restored to allow this function to continue processing in the context it was working in before the function call.

The last task a processor has to accomplish in order to perform a complete function call is to transfer the parameters to the function. It has to transfer the input parameters when the call is performed, and transfer back the return values after the execution of the function is complete. Again, this is most commonly done by storing these values in the stack, but it can also be done by using special registers inside the processor.

These four important steps: calling the function, saving and restoring registers, transfer of parameters and return of outputs can be done in different ways. It can even be done in different ways on the same processor. Normally it is the job of the compiler to produce machine code instructions that perform these three steps in a suitable way.

In order to make it possible to link object code from different compilers and to increase portability, a number of standards, recommendations and conventions have emerged. There is for example the calling convention whereby it is up to the calling function to save the registers or whereby it is up to the called function to save the registers. In the SPARC architecture, it is the called function that saves the callers registers. In a similar manner there are conventions for how parameters should be transferred.

When working with machine learning at the binary level, it is sometimes allowable to be more free, and for example not always save all registers if it is known that the called function will not use them. This not only provides a more efficient implementation, but also presents the opportunity to use special features of the architecture that are hard to express in a high level language.

As described above, the structure of a function on the machine code level can be considered as an array of integers divided having four parts:

Header
Body
Footer
Return

The header of a function does one or more of the three steps mentioned above. In the SPARC architecture, the header of the function saves the registers of the calling function and sometimes also transfers parameters. Which of the three steps the header does is different from processor to processor and from compiler to compiler.

Although most function structures need some kind of header to perform a function call, some special kinds of functions do not need a header, such as the leaf procedure in the SPARC architecture.

The header is fairly constant and normally does not have to be manipulated by the machine learning part of the program. It can be defined at an early stage, for example in an initialization phase of the system.

The body of the function does the actual work that the function is supposed to carry out. When the body of the function is entered, all of its parameters are accessible to it, and the registers from the calling function are saved. The body can then use any of the arithmetic instructions or call another function to compute the desired function.

The footer contains the "cleanup" instructions as described elsewhere.

A return instruction must always follows the footer. This instruction finds out where the call to this function was made from, and then jumps back to this location. The address to jump back to is either stored on the stack or in special registers in the processor.

SPARC Architecture

A system that directly manipulates machine code is not easily made portable, so it is recommended to choose one platform as a base for experimentation. The preferred system includes the SPARC architecture and SUN workstations. The reason for this is that is one of the most widely used architectures in the research community, with a stable UNIX operating system. It is also a relatively fast RISC architecture.

SPARC is an open architecture with several different implementations and manufacturers. The SPARC International, Inc. is a non-profit consortium for evolution and standardization of this architecture. A well known user of SPARC is SUN Microsystems Inc., which uses the architecture in all of its modern workstations.

SPARC Registers

As described above with reference to FIG. 3, the most important of the different kinds of registers as applicable to the present invention are the "windowed registers". It is between the windowed registers that almost all arithmetic and logic operators take place. There are a dozen other registers that are more important to the system software than to a client

50

application. The program counter is also an important register.

The windowed registers are divided into four classes:
Output registers O_0 to O_7
Input registers I_0 to I_7
Local registers L_0 to L_7

There are also eight global registers G_0 to G_7.

There are eight registers in each of these classes. When a save instruction is executed, it copies the contents of the output registers into a new set or bank of corresponding input registers. Register O_0 is copied into I_0, O_1 is copied into I_1, etc. The input register used is a new input register owned by the called function.

The values in the old input registers are kept or saved. It will be noted that the contents are not really copied. In reality these registers are the same, and only a pointer to the current set of register in incremented.

The processor has an internal storage for a few banks or sets of register like this (seven in the SPARC architecture), and if this limit is exceeded, the hardware and system software will save the contents in memory. For the user, this banked register mechanism can be thought of as a small internal stack.

It is important to note that a save instruction copies the contents of the calling function's output registers into the called function's input registers, while the restore instruction copies the contents of the returning function's input registers into the calling function's output registers.

When a function wants to call another function and pass some variables, it thus places the parameters in its output registers and makes the call. After the function call has been completed, the returning values can be found again in the output registers.

The local registers are local to the function working for the moment, and are used for temporary storage. A fresh set of local registers is provided when a new function is called, but there is no special transfer of values into these registers.

The global registers are always the same. They keep their meaning and content across function calls, and can thus be used to store global values. There are no alternative sets of this registers like there are with the input, output, and local registers.

Register Implementation

Some of the registers have a reserved usage. In some cases the reserved meaning is due to hardware, but in most cases the origin of these constraints are software conventions, as specified by the SUN Application Binary Interface (ABI).

In accordance with the present invention, almost all of the actual algorithm is written in a high level language like C. This prevents interference with the work done by the compiler. If this goal is abandoned, and the complete system were written in assembly language, even higher flexibility could be achieved. However, all portability would be lost, and it is therefore preferred to provide the manipulating program in a high level language such as C.

The global registers are preferably not used by the program, because these registers are likely to be used by the code generated by the compiler. Global storage for function structures can be provided in memory.

Global register zero has a special meaning. It is not a register where values can be stored. An attempt to read global register zero, always returns the value zero, and an attempt to write to global register zero does not change anything. This register is used when a zero constant is needed or when the result of an operation should be discarded.

Global register one is by convention assumed to be destroyed across function calls, so this register can be used by the functions that are manipulated by the machine learning algorithm.

Registers I_6, I_7, O_6, and O_7 are by convention used to store stack and frame pointers as well as the return address in a function call, so these should not be used by the program.

Local registers zero and one have special uses during an

4-2-

interrupt but this will not affect the functions.

Thus, the registers which are available for use in the SPARC architecture are global register G_1, input registers I_0 to I_5, output registers O_0 to O_5, and local registers L_0 to L_7.

SPARC Instructions

SPARC is a RISC architecture with a word length of 32 bits. All of the instructions have this size. Basically there are three different formats of instructions, defining the meaning of the 32 bits. The processor distinguishes between the formats by looking at the last two bits, bit 30 and bit 31. The three formats are:

CALL instruction

Branches, etc.

Arithmetic and logic instructions

In the CALL instruction, bit30 is one and bit31 is zero. The rest of the bits are interpreted as a constant that is added to the program counter (PC). The return address is stored in output register I_7.

Branches are mostly used for conditional jumps. They look at the last performed arithmetic operation, and if it fulfills a certain criteria, for example if the result is zero, then the program counter is incremented or decremented by the value of the last 22 bits in the instructions. In a branch, both bit 30 and bit 31 are zero.

The last group of instructions are the arithmetic and logic instructions between registers. These groups of instructions have bit 31 as one and bit 30 as zero. These are the preferred instructions for practicing the present invention. These instructions perform, for example, multiplication, division, subtraction, addition, AND, OR, NOT, XOR and different SHIFT instructions.

The arithmetic instructions can also be used as jump, call, and return instructions, if the result of the operation is put into the program counter register. In this way it is possible to jump to the address pointed to by the contents of a register.

When the current value of the program counter is saved in another register (out7), these instructions can be used as call instructions.

Call is equivalent to jmpl register,%o7, where jmpl means "jump long"; jump to the content of "register" and place the current program counter in register O_7.

Return (ret) from a normal procedure is equivalent to jmpl.

Return from a leaf procedure (ret) is equivalent with jmpl.

The constant eight causes the control to jump back past the original call and past this call's delay slot.

The return instruction puts the value of O_7 with a constant added thereto into the program counter, and causes the execution of the program to return.

There are also instructions for loading and saving information from the memory.

Control transfer instructions (CTI) like jump, call, and return, are somewhat special in the SPARC architecture. The instruction immediately after the jump instruction is executed during the transfer of control, or it could be said to be executed before the jump.

A SPARC assembly code listing can appear misleading because of the change in execution order, because a NOP instruction is placed after the CTI. The instruction after the CTI is say to be in the delay slot of the CTI. The reason for this somewhat awkward mechanism is that if the delay slot can be filled with a useful instruction, it will make the execution of the overall program more effective, because the processor can do more instructions in the same cycles. It is the hardware construction of the processor the makes this arrangement necessary.

The CTI and the delay slot are important in implementing the CGPS. Special rules require that a CTI that has a CTI in its delay slot. This is called a Delayed Control Transfer Couple.

The execution of a DCTI couple is awkward. The first (and only the first) instruction of the function that the first CTI

54

is pointing to is executed, then directly the execution jumps to the function that the second CTI is pointing at. The instruction after the second CTI is not regarded as being in a delay slot, and it is executed when control is returned. This phenomena is important for the construction of subroutines and external functions.

Leaf Procedures

The above described properties of the call and save instructions make it possible to use two particular kinds of procedures.

The first kind of procedure is a full procedure that uses the save and restore instructions, and the procedure is consequently allowed to use and manipulate input, output and local registers. The save and restore functions, however, consume processor cycles. If there is room inside the processor, the time consumption is moderate, but if storage in memory is needed it will take many processor cycles.

The solution to this problem is to use leaf procedures. They are called leaf procedures because they cannot call another procedure, and therefore leaves in the procedure structure of a program.

A leaf procedure does not perform a save operation, and works with the same set of registers as the calling procedure. To avoid interference with the content of the calling procedure, it only manipulates the output registers, which are assumed to be destroyed by the compiler across function calls. One of the elegant consequences of this technique is that the calling procedure does not have to know what kind of procedure it is calling. This means that linking of procedures works normally.

The difference between a leaf procedure and a full procedure is that it only manipulates the output registers, does not use save or restore, and has a special return instruction that looks for the return address in output register O_7 seven instead of input register I_7.

These details originate mostly from software conventions implemented in compilers. When the underlying principles are known, it is possible to have many more combinations of behavior of procedures. In some of examples in CPGS, hybrids of the normal procedures and leaf procedures are used, because it is known how the registers are used in the calling procedures.

Although the floating point features of the computer architecture are not specifically addressed herein, it is possible to use binary manipulating algorithms for floating point processors and units. This makes the present invention applicable to new domains of problems.

Evolutionary Algorithms

Examples of genetic algorithms were present above with reference to FIGs. 10 to 15. A genetic algorithm, for example, is an algorithm based on the principle of natural selection. A set of potential solutions, a population, is measured against a fitness criteria, and through iterations is refined with mutation and recombination (crossover) operators. In the original genetic algorithm, the individuals in the populations consist of fixed length binary strings. The recombination operators used are the uniform and 2-point crossover operators.

In genetic programming the goal of the system is to evolve algorithms or programs in a given language. Most genetic programming systems use a tree structure for the representation of programs, as in the above referenced patent to Koza. The most commonly used tree representation form is the S-expressions used in the LISP-language.

The tree representation guarantees that the evolved programs will be syntactically correct after the genetic operators are applied. In the original model, the only genetic operator apart from selection is the subtree exchanging crossover. This crossover operator swaps two subtrees in the individuals that undergo crossover.

A middle way between these two representation forms is messy genetic algorithms, which has a freer form of representation where for example the loci of gene is not tied to its interpretation. Instead genes are tagged with a name to enable identification.

Evolution of Machine Code Structures and Content

A basic machine learning system was described above with reference to FIGs. 1 to 18. Our discussion of the prior system provides many of the basic details of the present invention and is incorporated here by reference. The algorithm evolves fixed length binary strings, and uses crossover and mutation operators. The crossover is prevented from damaging certain bit fields within the 32 bits of the instructions. This protection procedure guarantees that only valid machine code instructions are generated through crossover. The mutation operator has a similar protection feature.

The system described below is a more complete machine learning and induction system, capable of evolving Turing complete algorithms and machine code functions. The system provides additional advantages including the following.

Use of several machine registers
Dynamic allocation of memory, (no recompilation needed)
Variable length of programs.
Multiple input parameters to functions
Unlimited memory through indexed memory
Automatic evolution of subfunctions
If-then-else structures
Jumps
Use of loop structures including for, while, repeat
Recursion, direct and through subfunctions
Protected functions e.g. division
String functions, and list functions
Linking any C-function for use in the function set

These goals are met by a program written mostly in C, which uses unrestricted crossover at instruction boundaries. Unrestricted crossover means that the crossover acting on the strings of instructions should be able to work blindly without checking what kind of instructions are moved and where.

There are many advantages of this if the goal is met. The implementation is very efficient, because the algorithm will only consist of a loop moving a sequence of integers, something a computer is very good at. The implementation will be simple and easily extendable because there will be a minimum of interaction and interference between parts of the program. Equally important is the fact that the program will be more easily ported to different platforms because the architecture specific parts can be restricted to a minor part of the program, and the crossover mechanism does not have to be effected.

It is easy to find examples of instructions and combinations of instructions where these properties do not hold. The normal call instructions constrains an offset that is added to the program counter. If this instruction is moved to another memory position, the call will point to another memory location where there might not be anything like a proper function. The SPARC architecture does not have calls with absolute addresses, which would still work after crossover. Instead, a call to an address specified by a register is used. The value in the register will be the same even if the instruction is moved by crossover.

The same problem exists with a normal branch instruction. It is also defined by a constant added to the program counter, and it cannot either be moved without changing this constant. But a certain class of instructions are possible to move unrestrictedly, without any unwanted effects. Machine language is quite flexible, and this restriction of instruction use is possible without abandoning the goals of the system.

Crossover

The crossover operators (uniform and 2-point) were described in detail above. See Figure 22j.

Mutation

The mutation operator picks an instruction at random, and checks whether it has a constant part or if it is only an operation between registers. If it has a constant part, a bit in this constant is mutated and also potentially the source and destination registers of the operation. If the instruction does not have a constraint part, the instruction's type, source and destination registers are mutated. See Figure 22i.

Calls and jumps are not mutated other than that they may be swapped for other instructions.

Casting and Execution of an Array

In a computer, all data is represented as integer numbers. To make programming a more easy task there are more data types and structures in a high level program, for example: strings, pointers, characters bitmaps, etc.

But all of these structures are, as stated above, translated into integers. Sometimes the programmer wants to translate an object of a certain type into an object of another type. The term for this is casting, one type is cast into another type. This is often only an operation in the high level language, at the machine code level they might be represented by the same integer.

Speed Increase

The efficiency of the present invention can be 1,000 times faster than a system coded in an interpreting language, and there are reasons to believe that similar performance enhancements are possible for other kinds of meta manipulating programs. If, however, these speed enhancements still would not be enough there are a few ways to further improve performance.

The main system described here has a number of advanced features like subroutines, recursion, etc. If extremely fast performance is needed and if a system can do without these features and if the goal is to only evolve arithmetic and logic expressions of a number of "in" variables, then the efficiency can be increased even further.

More specifically, if it is acceptable to decompile the system between runs, for example if the problem parameters are

59

given beforehand, then execution speed can be increased further. Finally, there is the possibility of coding and optimizing the entire system in assembly language which has the fastest execution.

System Parameters

The following is a list and a brief description of the parameters that are used to control the present machine learning system.

Population size
Number of fitness cases to evaluate. Training will halt when the system has tried these many fitness cases.
Mutation probability
Crossover probability
Parsimony pressure start value
Number of incremental steps for parsimony pressure
Size of each incremental parsimony pressure step
Maximum program size
Initial mean program size
Random number generator seed
Success threshold for fitness
Maximum number of iterations, in recursion and loops
Maximum size of initial terminal set numbers
Number of bits to be mutated in terminal numbers
Number of input registers to use
Number of output registers to use
Number of external functions
Maximum number of automatic subfunctions
Flags determining which instructions to use: ADD, SUB, MUL, SLL, SRL, XOR, AND, OR

These parameters are read from an initialization file before each training session. All effects of these parameters are dynamically allocated, and there is no recompilation of the system necessary. This implies that a flexible system can be implemented on a small computer without an onboard compiler. These parameters would, of course, be different for other machine learning systems or for repeat calculation systems on run-time data.

Machine Code Instruction Implementation

The following SPARC machine code instructions are used in the CGPS implementation.

ADD, Addition

SUB, Subtraction

MUL, Integer multiplication

SLL, Shift left

SRL, Shift right

XOR, Exclusive or

AND, Logical And

OR, Logical Or

Call L_{0-7}

The arithmetic and logic instructions, all instructions except the last call instructions, come in four different classes. These arithmetic instructions can have the property of affecting a following if-then branch or not affect it. They can also have the property of being an operation between three registers, or an operation between two registers and a constant. The combinations of these two classes makes four different variants of the arithmetic instructions.

An arithmetic instruction could for example add output register O_1 with output register O_2 and store the result in output register O_3, or it could add a constant to output register O_2 and store the result in output register O_3.

The 32-bit instruction format has room for a constant of ±4196. In this manner, a single instruction is substantially equivalent to many elements in an ordinary machine learning system: one element for the operator, two elements for the two operands, and one element for the destination of the result. This approach is thus quite memory effective using only four bytes of memory to store four nodes.

There are eight different call instructions, one for each of the local registers. The call instruction jumps to an address given by the content of the local registers in the function. The local registers are thus reserved for this usage in our implementation, and they have to be initialized in the header of every function to make sure that they point to a function in use.

These instructions are the basis for external functions and subroutines. How many of these instructions are used in the initialization is determined by how many subroutines and external functions that are used.

The maximum number of external functions and subroutines are in this implementation limited to eight. If more external functions are needed, then another initialization technique can be used where the call instructions jumps to the address given by a constant added to a local register. The instruction format allows for storage of such constants within the instruction.

The division instruction can be used directly as a basic instruction. Protection from division by zero can be provided by catching the interrupt generated by hardware. It is, however, more effective to make protected division as an external function, with a few instructions checking for division by zero.

<u>Initialization</u>

Initialization is the first task the system performs when presented with a giving training situation. See FIGs. 22b, 22c. Initialization can be divided into four steps.

Allocation of memory for the population, etc.
Initialization of header
Initialization of footer
Initialization of the function body

<u>Initialization of Memory</u> FIGs. 22b, 22c.

The memory used for the individuals in the arrays is a linear array of integers. The array is divided into blocks determined by the system parameter maximum length. A fixed maximum length is thus reserved for every individual. If there

[illegible]

are subroutines, then this memory is allocated for every subroutine according to the maximal number of subroutines. The program and its subroutines then can vary in length within these boundaries.

The advantages with this paradigm include very simple memory management, without garbage collection. The approach with linear memory is efficient and natural for the use of binary code. There is also a constant -- easily calculated -- memory usage once the system is initialized. This can be a requirement in real-time systems, and in systems with limited memory resources.

An example population of one million individuals with no subroutines and 40 nodes maximum per individual in this way occupies 40MB of RAM memory, which is manageable on an ordinary workstation.

<u>Initialization of Header</u> FIGs. 22b, 22c.

The initialization of the header consists of two parts; one that is fixed, and one that depends on the number of subroutines and external functions.

The fixed part of the header is a NOP instruction and a save instruction.

The NOP instruction is needed for the rare case that this function is called from another function as a subroutine by a control transfer couple as described above. A control transfer couple can arise after an unrestricted crossover where two jumps or call instructions are placed after each other.

Only the first instruction of the first call is executed. If this first instruction is a save instruction, which is normal, then this save would be executed alone and the control will go to the address of the second call which probably will also be a save instruction.

The first of these two save instructions will be unbalanced, not corresponding to a restore, and the registers will be corrupt. If instead a NOP is placed in the header of every instruction, the NOP instruction can be executed safely without affecting the state of the machine.

This point is probably the most special case in the implementation, but it enables the use of unrestricted crossover. When an external function is to be used in the system, it should also have a NOP instruction or another harmless instruction as its first instruction.

The second part of the header initializes the local registers. The local registers are used to store jump addresses of subroutines and external functions. This part of the header contains load instructions that load the appropriate addresses into the local registers. The current addresses are put here in the header during initialization, and then when an individual function is executed, it first executes the header and thus puts the right values in the local registers, which guarantees that the later call and jumps will be performed to the desired addresses. This part of the header varies in sizes by the number of subroutines and external functions.

<u>Initialization of Footer</u> FIGs. 22b, 22c.

The initialization of the footer is simple. The footer consists of two fixed instructions; one NOP instruction followed by a restore instruction. The NOP instruction is used in the last instruction in the body as a control transfer instruction like a jump or a call. In this case, the NOP instruction goes into the delay slot of the jump instruction and is executed during this procedure call. If the NOP instruction was not present, the restore instruction or the return instruction that follows the footer would go into the delay slot which would corrupt the registers. We have previously discussed headers and that discussion applies here also.

<u>Initialization of Return Instruction</u>

A return instruction must follow the footer.

<u>Initialization of Function Body</u> FIGs. 22b, 22c, 22d.

The function body is initialized by for each memory cell by randomly selecting an instruction from the set of instructions that the user has put into the function set, including call instructions using local registers. If the picked instruction is an arithmetic instruction, input and output registers are chosen for operands and destination according to the parameters supplied by the user.

We have previously discussed headers, bodies, footers and return instructions, and that discussion applies here also.

With a certain probability, an instruction is given either a constant and register as operands, or two registers as operands. If one of the operands is a constant, this constant is randomly generated to a maximum size defined by its parameter and put into the instruction. The instruction has room for constants within the range of ±4196, within the 32 bits of the instruction.

Subroutines

Subroutines are modularisations within an individual that spontaneously change during evolution. A subroutine has the same structure as a function call, and consists of parameters transferred in the out variables, and a call function.

An individual in this system is a linear array of numbers. This array is divided into a number of pieces of uniform size. The number of pieces corresponds to the maximum number of the subroutine parameter. Every such memory section is a subroutine. A subroutine is organized the same as a main function, with a header, a footer, and a function body.

In the header of a subroutine, the local registers are initialized to contain the addresses of the other subroutines that can be called from this subroutine. When the recursion option of the system is switched off, the local registers are only loaded with the addresses of subroutines higher up in the hierarchy. So if the maximum number of subroutines is set to four, the local registers L_0 to L_3 in the main function are initialized with the addresses of subroutines 0 to 3, while the local registers L_0 and L_1 in the first subroutine are initialized with the addresses of subroutines 3 and 4.

Remaining unused local registers are initialized with the value of a dummy function that executes a return when called. With the scheme it is possible to allow unrestricted crossover between individuals and between subfunctions, because the local registers will always be initialized in the header of each subfunction to a correct address. The "call local register" instructions can thus be freely copied in the population.

Recursion

Recursion can be implemented by initializing the local registers not only to the values of subroutines higher up in the hierarchy, but also to the current subroutine itself. Recursion can also be implemented in a small external function, a leaf function. The difference between the two approaches is small, but the main advantage of the later method is that the header can be kept the same regardless of if recursion is used or not, which makes the implementation less complex.

Regardless of which approach is used for recursion, there will always be the problem of infinite chains of instruction calls. The halting problem makes it impossible to know in advance which recursive functions will stop and which will not.

The solution to this is to have a global variable in a memory location that is incremented every time a function is called. If a certain limit is reached, the execution of the individual is abandoned. The code for this checking is placed in the header of each function if the first recursion method is used. If the second method is used, this code is the first part of the recursion leaf procedure.

Leaf Procedures as Program Primitives.

Loops are implemented in a way similar to recursion. A leaf procedure is used which performs a test of a variable. Depending on the outcome of the test, a loop branch is either performed or not.

The test can be whether the last performed arithmetic

[illegible]

instruction produced zero as a result. This is accomplished by checking the so called zero flag in the processor. Out loop structures can be used simultaneously by checking other integer conditions from the last instruction.

This branch is made with a return instruction, which is a "longjump" instruction jumping to the address of a register with a constant added thereto. This constant can be positive or negative. The normal return instruction jumps back to the address given by the content of register O_7 or I_7 incremented by eight. These eight bytes cause the return to skip the original call instruction and its delay slot.

When this function is used in a loop, the constant is made negative so that return jumps back to an address before the call to the "loop leaf procedure". The fact that the different call addresses so far are in the execution of the individual, all are stored and accessible to the leaf procedure which makes it possible to construct a large number of control structures. With this method, it is possible to define efficient external leaf procedures that implement ordinary jumps, all kinds of loops, indexed memory, if-then-else structures, and protected functions.

A limit of eight external functions is implied by the number of local registers. This is enough for many applications, but if other domains demand more external functions and subroutines, it is possible to define their addresses as constants added to local register. This technique provides the possibility of using an almost unlimited number of primitives, external function and subroutines.

External Functions

It is possible to incorporate any C-function into the function set. Any C-module can be compiled and linked into the system. There are a number of steps that must be taken in order to make this linking successful.

First the module could be compiled to assembler code by the "-S" flag of the "cc" compiler. A NOP operation is then added before the other instructions in the function. The name of the function is added in a call array in the main C kernel code, and potentially a string name is added to an array for disassembling. After recompilation the system is ready to use the new external function. This approach can be extended and made more automatic with dynamic linking, etc.

C-Language Output

A disassembler can be provided which translates the generated binary machine code into C-language modules. This disassembler feature is provided with the goal that the output from the system should be able to be used directly as a C-module, and it should be possible to compile and link it to another c-program. With this feature the system can be used as a utility to a conventional development environment. The disassembler could also be requested to produce assembler code.

Portability Methods

The main disadvantage with the machine code manipulating technique is that it is machine specific. There are, however, a number of methods to make the system more portable.

The first is to allow for unrestricted crossover. This will make the machine independent part of the code quite small, only defining a number corresponding to instructions, etc. In this way it is possible to write a system that runs on multiple platforms without changes in the code.

An more radical approach is to let the system look for itself in its binary code how it is translated to machine language. Many template functions exist in the C-code with translations to machine code that define how the processor is working. This method is only possible for compiling systems that do simple functions with arithmetic, without jumps, subroutines, etc.

Another approach is to lean more on the hardware interrupts for portability. For example, every time an illegal instruction is encountered by the processor, a hardware interrupt is generated. Using the interrupt features in Unix it is possible to be less restrictive when manipulating binaries knowing that the system will catch some of the erroneous structures.

This increases portability, because faults can sometimes be ignored that arise due to incompatibilities of different architectures. In an extreme example, a machine learning system can be implemented by using mutation and crossover at the bit level where all of the many illegal situation are caught by the processor memory management hardware or file system protection mechanisms.

The ideal portability situation would be to have a special language, present on different platforms, for this kind of run-time binary manipulation.

Using Tree Representation

In the preferred implementation of the present invention, a binary string approach is used for representation of individuals. Binary strings are the common representation form of genetic algorithms while tree representations are used in conventional genetic programming.

There may be applications where tree representation and crossover at a tree level is interesting to study and consequently it would be interesting to have the individual in a tree form. This can be accomplished in several ways.

One way is to use two different NOP instruction as parenthesis in the code. The crossover operator then scans through the code, matches these parenthesis, and performs tree crossover accordingly. This could potentially make the crossover operator less efficient for large individual sizes.

Another way is to break up the individual into segments in the memory, where part of the tree is presented as jump statements to subtrees. This would still make it possible to execute the individual directly. The crossover operator here must also scan through the individual, but it only has to move smaller blocks of code that could be allocated dynamically.

The third way is to let every individual have an extra array associated with it, that carries information about where subtrees start and stop. This array will not be involved in the execution of the individual, and it will only be used by the crossover operator to locate subtrees.

Applications

The present invention can be applied to any problem in which a computer algorithm manipulates a structure that later should be interpreted as instructions. Examples of these kind of such applications include the following.

Genetic algorithms and genetic programming
Cellular automata
Artificial life systems like Tierra
Rule induction systems
Decision trees
LISP or PROLOG interpreters

The invention is especially suited for applications within areas that require:

High execution speed
Real time learning
Large memory structures
Low end architectures, e.g. consumer electronics
Well defined memory behavior

Example

FIGs. 19 to 21 illustrate a function structure and associated registers in accordance with the Turing complete machine learning system of the present invention.

FIG. 19 illustrates an array of functions F_0 to F_6, each of which consists of a main function MAIN and two subroutines SUB1 and SUB2. Each function has a maximum, but variable length. Portions of the functions occupied by the header are indicated as H, the instruction body as B, the footer as F, and the return instruction as R.

It will be noted that the main functions and the subroutines SUB1 and SUB2 also have variable, but maximum

10

lengths. Any memory not occupied by an operative portion of a function is unused.

FIG. 20 illustrates a function FUNCTION which consists of a main function MAIN, and two subroutines SUB1 and SUB2. Further illustrated are the input registers, output registers and local registers of BANK0 which is used by the main function, a leaf function, and a dummy function. The latter functions are stored in the memory 12.

The starting addresses of the functions MAIN, SUB1 and SUB2 are designated as ad0, ad1, and ad2, whereas the starting addresses of the functions LEAF and DUMMY are designated as ad3 and ad4 respectively. The instructions which can be placed in the functions MAIN, SUB1 AND SUB2 and which are subject to alteration for the purpose of machine learning are limited to those which branch to the addresses ad0 to ad4.

The function LEAF performs the operation of protected division, where a variable "c" stored in the input register I_2 is to be divided by a variable "b" stored in the input register I_1 . More specifically, an instruction "TEST I_0" tests whether or not the variable "b" is zero. If so, the result of the division would be an infinitely large number, which constitutes an error for the system.

If the test instruction detects that the value of I_1 is zero, the next instruction is skipped and the next instruction, which is, a RETURN instruction, is executed, returning control the calling function.

It the test instruction detects that the value of I_1 is not zero, the division operation c/b is performed, the result stored in the input register I_3 (as a variable d), and control returned to the calling function.

The function DUMMY consists of a return instruction that merely returns control to the calling function.

The headers of the functions MAIN, SUB1 AND SUB2 each include a SAVE instruction, and three instructions that initialize the local registers with the addresses of functions that can be called by the particular function. The SAVE

F71

instruction causes the contents of the output registers of a calling function to be copied into the input registers of the called function as described above.

In the function MAIN, these initialization instructions cause the addresses ad1, ad2 and ad3 to be stored in the local registers L_0, L_1, and L_3 respectively.

In the normal hierarchial arrangement of the invention, functions are only allowed to call functions which are lower on the hierarchy, and leaf functions. Thus, the function SUB1 is allowed to call the function SUB2, but not the function MAIN. This is accomplished by storing ad2 for the function SUB2 in L_0, ad3 for the function LEAF in L_1, and ad4 for the function DUMMY in L_2.

The function SUB2 is only allowed to call leaf functions. Therefore, the address ad3 for the function LEAF is stored, in L_0, and the address ad4 for the function DUMMY is stored in L_1 and L_2.

FIG. 20 further illustrates how an arithmetic instruction is performed by executing a function, and how a variable is passed from a calling function to a called function.

More specifically, execution of the instruction $I_0 = I_1*2$ by the function MAIN causes the variable "d" in the register I_1 to be multiplied by 2, and the result stored in the register I_0 as indicated by an arrow. The contents of the register I_0 are indicated as a=b*2.

To pass the value of "a" (in I_0) to a called function, an instruction $O_0=I_0$. This causes the contents of the input register I_0 to be copied into the output register O_0. The next instruction, CALL L_0, causes control to be transferred to the function having the starting address ad1, which is the function SUB1. The call instruction causes the contents of the output registers of the function MAIN to be copied into the input registers of the function SUB1, and thereby pass the value of the variable "a" which was stored in O_0 of the function MAIN to the input register I_0 of the function SUB1.

Upon return from the call to SUB1, the contents of the input registers of the function SUB1 are copied to the output registers of the function MAIN. The next instruction in the function MAIN is $I_0=O_0$. This causes the contents of O_0 to be copied to I_0, and thereby pass the variable that was generated by the function SUB1 and stored in I_0 thereof to the function MAIN.

FIG. 21 is similar to FIG. 20, but illustrates an arrangement including one subroutine SUB1 and one leaf function, with the subroutine SUB1 being allowed to perform recursion. In this example, the function SUB1 can call the function MAIN, itself, and the function LEAF.

This is accomplished by storing ado (the address of the function MAIN) in L_0, storing ad1 (the address of itself) in L_1, and ad2 (the address of the function LEAF) in L_2.

A detailed flowchart of the Turing complete machine learning system is illustrated as a flowchart in FIGs. 22a to 22k.

FIG. 22a is main diagram of the system.

FIG. 22b illustrates the details of a block SETUP in FIG. 22a.

FIG. 22c illustrates the details of a block INITIALIZATION in FIG. 22a.

FIG. 22d illustrates the details of a block INITIALIZE LOCAL REGISTERS IN HEADER in FIG. 22c.

FIG. 22e illustrates the details of a block CREATE INSTRUCTION in FIGs. 22c and 22i;

FIG. 22f illustrates the details of a block MAIN CGPS LOOP in FIG. 22a.

FIG. 22g illustrates the details of a block CALCULATE INDIV[N] FITNESS in FIG. 22f. The steps indicated by a left bracket are performed by a C-instruction "Predicted_Output[i]=((function_ptr) Indiv[n]0(input[1]...Input[k])".

FIG. 22h illustrates the details of a block PERFORM GENETIC OPERATIONS in FIG. 22f.

FIG. 22i illustrates the details of a blocks MUTATE

INDIV[1] and MUTATE INDIV[3] in FIG. 22h.

FIG. 22j illustrates the details of a block CROSSOVER INDIVS [1] and [3] in FIG. 22h.

FIG. 22k illustrates the details of a block DECOMPILE CHOSEN SOLUTION in FIG. 22a.

Sub a5 ~~The following pages of this specification include articles entitled "COMPLEXITY COMPRESSION AND EVOLUTION", "GENETIC PROGRAMMING CONTROLLING A MINIATURE ROBOT", PROGRAMMIC COMPRESSION OF IMAGES AND SOUND", AND GENETIC REASONING EVOLVING PROOFS WITH GENETIC SEARCH". These articles constitute part of description of the present invention as disclosed herein.~~

In summary, the present invention overcomes the drawbacks of the prior art by eliminating all compiling, interpreting or other steps that are required to convert a high level programming language instruction such as a LISP S-expression into machine code prior to execution or that are required to access Learned Elements or run-time data in data structures.

This makes possible the practical solutions to problems which could not heretofore be solved due to excessive computation time. For example, a solution to a difficult problem can be produced by the present system in hours, whereas a comparable solution might take years using conventional techniques.

Various modifications will become possible for those skilled in the art after receiving the teachings of the present disclosure without departing from the scope thereof.

Complexity Compression and Evolution

Peter Nordin
Universität Dortmund
Fachbereich Informatik
Lehrstuhl für Systemanalyse
D-44221 Dortmund
nordin@ls11.informatik.uni-dortmund.de

Wolfgang Banzhaf
Universität Dortmund
Fachbereich Informatik
Lehrstuhl für Systemanalyse
D-44221 Dortmund
banzhaf@ls11.informatik.uni-dortmund.de

Abstract

Compression of information is an important concept in the theory of learning. We argue for the hypothesis that there is an inherent compression pressure towards short, elegant and general solutions in a genetic programming system and other variable length evolutionary algorithms. This pressure becomes visible if the size or complexity of solutions are measured without non-effective code segments called introns. The built in parsimony pressure effects complex fitness functions, crossover probability, generality, maximum depth or length of solutions, explicit parsimony, granularity of fitness function, initialization depth or length, and modularization. Some of these effects are positive and some are negative. In this work we provide a basis for an analysis of these effects and suggestions to overcome the negative implications in order to obtain the balance needed for successful evolution. An empirical investigation that supports our hypothesis is also presented.

1 Introduction

The principle of Occam's Razor, formulated 700 years ago, states that from two possible solutions to a problem we should choose the shorter one. Bertrand Russell claims that the actual phrase used by William of Ockham was: "It is vain to do with more what can be done with fewer". A famous example of Occam's Razor is when the Polish astronomer Copernicus argued in favor of the fact that the earth moves around the sun and not vice versa, because it would make his equations simpler. Many great scientists have formulated their own versions of Occam's Razor. Newton, in his preface to Principia, preferred to put it as; "Natura enim simplex est, et rerum causis superfluis non luxuriat". (Nature is pleased with simplicity, and affects not the pomp of superfluous causes.) The essence of Occam's Razor is that a shorter solution is a more generic solution. The process of inferring a general law from a set of data can be viewed as an attempt to compress the observed data. Some researchers have claimed that this principle could be the basis of many cognitive processes in the brain (Wolff 1993).

In this paper, we argue that one of the foundations of Evolutionary Algorithms in general, and Genetic Programming in particular, is that they have the built in property of favoring short solutions and sub-solutions. This property might be one of the reasons that Evolutionary Algorithms work so efficiently and robustly in a diverse set of domains. The compression property could also be responsible for the ability of a solution to be generic and applicable on a larger set of data than the set of data or fitness cases used during evolution. The other side of the coin is that the built in compression pressure in certain cases is too strong and results in premature convergence and failure to adapt to complex fitness functions. The Evolutionary Algorithm could choose a short but incomplete solution instead of a long but complete solution. The strength of the pressure is dependent on the different attributes of a particular Evolutionary Algorithm such as, representation, genetic operators and probability parameters.

The bottom line is that it is helpful to be aware of this compression pressure and to try to keep it on a balanced optimum level during evolution.

In this paper, we focus on the problem of symbolic regression of programs with a genetic algorithm. We have used a variant of a variable length genetic algorithm operating on a string of bits to evolve an algorithm or program for a register machine (Nordin 1994). Using a register machine makes the analysis of introns more straight forward, and using a bit string representation will simplify the complexity reasoning. The argumentation, however, is analogous for standard tree-representation Genetic Programming and the reasoning is useful for other evolutionary systems.

1.1 Destructive Crossover

We would like to start by defining some important concepts. These are destructive crossover, different kinds of introns and effective complexity.
A crossover acting on one block or segment of the code in an individual, might have different results. In one extreme case the two blocks that are exchanged in crossover are identical, therefore, the performance of the program is not affected at all. Normally, however, there is a high probability that the function of the program is severely damaged, resulting in a fitness decrease for the individual. In Figure 1 we can see a typical distribution of the effect of crossover on fitness in an early generation of the symbolic regression problem from section 3. The x-axis gives the change in fitness $\Delta f_{percent}$ after crossover f_{after}.($f_{best} = 0$, $f_{worst} = \infty$).

$$\Delta f_{percent} = \frac{f_{before} - f_{after}}{f_{before}} \cdot 100 \qquad (1)$$

Individuals with a fitness decrease of more than 100 percent are accumulated at the left side of the diagram. This diagram shows that the most common effect of crossover is a much worsened fitness (the spike at the left). The second most common effect is that nothing happens (the spike of zero). Below we use the term "probability of destructive crossover" for the probability that a crossover in the program or block will lead to a deteriorated fitness value, comprising the area left of zero in Figure 1, $p_d = P(\Delta f_{percent} < 0)$. The term could be used for complete programs as well as for blocks in programs.

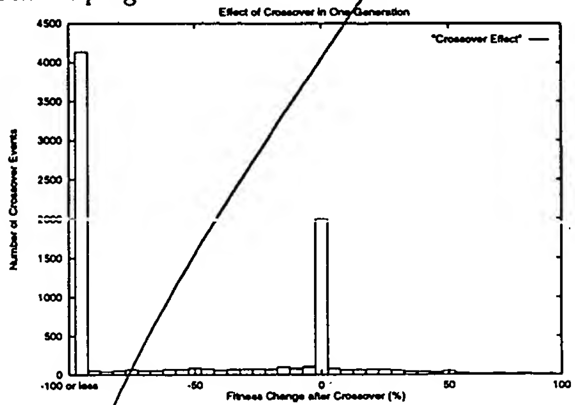

Figure 1: Effects of Crossover in one Generation.

1.2 Introns

As a rule, a solution evolved by Genetic Programming systems contains segments of code that do not seem to perform any useful functions, seem to be unnecessarily lengthy or that are not executed at all within the program. Similar redundant structures are present in nature within DNA and are called "introns" (Watson 1987). The idea of explicitly inserting introns in Genetic Algorithms has previously been investigated (Levenick 1991, Forrest 1992).

In the experiments performed in Section 3 below we use the term *intron segment* for a block that does not affect the behavior of the entire program for any of the fitness cases. The code is thus neutral on the phenotype level. Notice however that an intron segment may affect the output of a program for other inputs than the current ones. By the term *block* we mean any subset of the code regardless of its representation. It could be a sequence of binary digits in a binary string, a structure in a messy genetic algorithm, or a set of nodes within a tree representation. An *active block* is a block of code that is *not* an intron block and *does affect* the properties on the phenotype level.

Examples of introns can be found in most unedited individuals from a genetic programming run. The system can be very creative in finding such blocks. Some typical examples in S-expression notation are:

(NOT (NOT X)), (AND ... (OR X X)), (+ ... (- X X)), (* ... (DIV X X)), (MOVE-LEFT MOVE-RIGHT), (IF (2 = 1) ... X), (SET A A)

In reality, neither the extent to which a block affects output, nor the sensitivity of crossover is a discrete property of the block. There is a scale or distribution, both on, how sensitive the block is for crossover and how much it affects the program output. To simplify terms, we define an *absolute intron* as a block of code that neither affects the output of the program nor is sensitive to crossover. A crossover inside such a block will not affect the performance of the program. The following is an example of an absolute intron structure that can evolve when using the `if` function:

```
if 2<1 then {Absolute Intron Block ...}
       else {Active Block ...}
```

We call an intron *global* if it is an intron for every valid input to the program, and we call it *local* if it acts as an intron only for the current fitness cases and not necessarily for other valid inputs. This distinction is important for the generalization capabilities of a program, see Section 2.4.

In Section 3, we introduce a method for measuring the size of the intron segments in Evolutionary Algorithms. We look at intron blocks that do not affect the behavior of the individual for any of the fitness cases. This kind of intron will not be totally immune against crossover but, as long as the population contains some of these segments, swapping two of them by crossover will not affect the performance of the two individuals involved. Introns of this kind arise by a mutual agreement among the individuals to keep these sort of NoOperation code blocks.

1.3 Effective Complexity

By *complexity* of a program or program block we mean the length or size of the program measured with a method that is natural for a particular representation. For a tree representation, this could be the number of nodes in the block. For the binary string representation, in our work it is the number of bits, etc. The absolute length or *absolute complexity* is the total size of the program or block. The effective length or *effective complexity* of a block, or program, is the length of the active parts of the code within the program or block, in contrast to the intron parts.

2 Program Complexity, Effective Fitness and Evolution

Genetic programs do not seem to favor parsimony in the sense that the evolved program structures become short and elegant measured with the absolute size of an individual (Koza 1992). Instead, evolved programs seem to contain a lot of garbage and the solutions do not give an elegant impression when first examined. On the contrary, solutions look unnecessarily long and complex.

In this section, we give a reason for which a program has the tendency of increasing its absolute length during the course of evolution and at the same time favoring Parsimony. The crucial point is to measure *effective length* instead of *absolute length*. Observing the effective length will clearly show that the genetic programming system not only favors parsimonious solutions for the final result, but constantly, for sub-solutions during the evolution of the population.

Let us say that we have a simple GP system with fitness proportional selection and crossover as genetic operators. The crossover operator could be any crossover operator exchanging blocks of code such as the standard tree based subtree exchanging crossover (Koza 1992) or two point bit string crossover (Nordin 1994). If we have an individual program with a high relative fitness in the population, it will be reproduced according to its fitness by the selection operator. Some of these new copies will undergo crossover and will loose one block and gain another. If the crossover interferes with a block that is doing something useful in the program, then there is a probability that this new segment will damage the function of the block, see Figure 1. In most cases, the probability of damaging the program is much greater than the probability of improving the function of the block. If, on the other hand, the crossover takes place at a position within an absolute intron block, then by definition there will be no harm done to this block or to the program. A program with a low ratio of effective complexity to absolute complexity has a small "target area" for destructive crossover and a higher probability to constitute a greater proportion of the next population.

The ability to create and add introns during evolution is another important property of the system and its primitives. This ability could depend on parameters like initial individual size, function set, and fitness function. The additions of introns could be viewed as a way for the program to self-regulate the crossover probability parameter or as a "defense against crossover" (Altenberg 1994).

We can formulate an equation with resemblance to the Schema Theorem (Holland 1975) for the relationship between the entities described above. Let C_{ej} be the effective complexity of program j, and C_{aj} its absolute complexity. Let p_c be the standard genetic programming parameter giving the probability of crossover at the individual level. The probability that a crossover in an *active block* of program j will lead to a worse fitness for the individual is the probability of destructive crossover, p_{dj}. By definition p_{dj} of an absolute intron is zero. Let f_j be the fitness[1] of the the individual and $\overline{f^t}$ be the average fitness of the population in the current generation. If we use fitness proportionate selection[2] and block exchange crossover, then for any program j, the average proportion P_j^{t+1} of this program in the next generation is:

$$P_j^{t+1} \approx P_j^t \cdot \frac{f_j}{\overline{f^t}} \cdot \left(1 - p_c \cdot \frac{C_{ej}}{C_{aj}} \cdot p_{dj}\right) \qquad (2)$$

In short, Equation (2) states that the proportion of copies of a program in the next generation is the proportion produced by the selection operator less the proportion of programs destroyed by crossover. Some of the individuals counted in P_j^{t+1} might be modified by a crossover in the absolute intron part, but they

[1]Notice that this is not standardized fitness used in GP. Here a better fitness gives a higher fitness value (GA).

[2]The reasoning is analogous for many other selection methods including more elitist strategies

are included because they still show the same behavior at the phenotype level. The proportion P_j^{t+1} is a conservative measure because the individual j might be recreated by crossover with other individuals, etc.[3] Equation (2) could be rewritten as:

$$P_j^{t+1} \approx \left(\frac{f_j - p_c \cdot f_j \cdot C_{ej} \cdot \frac{1}{C_{aj}} \cdot p_{dj}}{\overline{f^t}} \right) \cdot P_j^t \quad (3)$$

Here we see that we can interpret the crossover related term as a direct subtraction from the fitness in an expression for reproduction through selection. In other words, reproduction by selection and crossover acts as reproduction *by selection only*, if the fitness is adjusted by the term:

$$-p_c \cdot f_j \cdot C_{ej} \cdot \frac{1}{C_{aj}} \cdot p_{dj} \quad (4)$$

This could thus be interpreted as if there where a term (4) in our fitness proportional to program complexity.

We now define "effective fitness" f_{ej} as:

$$f_{ej} = f_j - p_c \cdot f_j \cdot C_{ej} \cdot \frac{1}{C_{aj}} \cdot p_{dj} \quad (5)$$

It is the *effective fitness* that determines the number of individuals of a certain kind in the next generation.

Under these assumptions, it is possible to show that an individual can win a higher proportion of the next generation in spite of the fact that it has worse fitness. By loosing a block with high effective complexity with a moderate contribution to fitness it can increase its chances of survival. The individual is then trading normal fitness for effective fitness by reducing its complexity and a less fit individual can take a higher proportion of the next generation. This is an undesired phenomenon in most training situations.

2.1 Crossover Protection

An individual can do a number of things to protect itself from crossover. As discussed above, it can increase the absolute complexity by adding introns. The possibility of adding introns is limited by a number of factors for any given GP system under evolution. A system cannot add introns with arbitrary complexity instantly.

[3] The event of recreating individuals can be measured to be low except when applying a very high external parsimony pressure which forces the population to collapse into a population of short building blocks.

One other possibility for the individual to protect itself is to reduce the effective length by finding a more parsimonious solution, but this ability is limited by the fitness function and the dynamics of the system. It is in the balance between these two strategies that the parsimony pressure of a GP system appears.

If the effective length cannot be further reduced then the system will try to add more introns. This is on condition that the maximal length of depth of this system is not exceeded. If sufficient intron adding is allowed by the dynamics of the system, then it can gradually balance out differences in the effective fitness' by a short and a long program. All genetic programming systems have some defined maximal size of the program structure. This size is important because chosen too small it may limit the additions of intron, thus preventing the system from finding a perfect solution. In certain cases it is necessary for the intron blocks to be many times the size of useful blocks before an exact solution can be found. This means that it is important to adjust the allowed maximum size of individuals according to the composition of the fitness function.

A third possible way to reduce the probability for destructive crossover is to allow crossover at certain points only. Where crossover should be allowed could also be evolved through a suitable representation in the individual of allowed crossover points. This is common in nature, where there are numerous sophisticated systems for protection from changes in genetic material. Sexual recombination in higher species in particular is only allowed at very well defined points (Watson 1987).

The composition of the fitness function influences the sensitivity of the system too. For example, by altering the size of constants and scaling of different contributing part of the fitness function, it could be possible to reduce the crossover sensibility below the threshold of interference with the perfect solution.

2.2 Balanced Evolution

Adding an external parsimony pressure is one way to balance up the effective fitness. This pressure P is in its simplest form proportional to the absolute length of the program and subtracted from the fitness expression. This results in the following equation for the effective fitness:

$$f_{ej} = f_j - P \cdot C_{aj} - p_c \cdot f_j \cdot C_{ej} \cdot \frac{1}{C_{aj}} \cdot p_{dj} \quad (6)$$

The external parsimony pressure could be made variable during evolution and could also be allowed to have a negative value if necessary.

When evolving functions with non-continuous fitness functions for instance where the results is an integer, it is important to scale the output from the fitness function so that the smallest change in fitness, the granularity, is balanced against the biggest change in complexity. Otherwise, the change in fitness can drown in the change of complexity, see equation 5.

When heterogeneous complex fitness functions are used, it is also important to balance the contribution from the different parts in the fitness function.

The average initial size or complexity will affect the average complexity pressure in the beginning of a genetic programming session. The complexity pressure is proportional to the relation between the effective and absolute complexity. A high complexity at the initial state will give a low pressure in the beginning which is of importance for the dynamics of the whole training session.

Other strategies that could help evolving programs to obtain a shorter effective length are the introduction of control structures like loops or the use of subfunctions and other modularization techniques. In the next section we give a brief motivation for spontaneously emerging subfunctions.

2.3 Spontaneously Emerging Subfunctions

A natural tool for humans when defining an algorithm or computer program is to use modularization and divide the solution into smaller blocks of code. Different modularization techniques have been suggested for use with genetic programming, where the most thoroughly evaluated are automatically defined functions (ADF) (Koza 1994). Other examples of modularization techniques are Module Acquisition (Angeline 1993) and The Encapsulation Operation (Koza 1992). All modularization techniques are ways of encapsulating blocks of code. ADFs encapsulate blocks that becomes subroutines which could be called from the main program or from another subroutine. A subroutine can be called more than once from the same program. This means that a program can reduce its effective length by putting frequently used identical blocks into a subroutine. As we have seen, a small effective length increases the chances of survival of this offspring. This is in our mind one of the main reasons for which ADFs and other encapsulation techniques work spontaneously.

If a block of code with effective complexity δC_{block} within an individual is present n times, then the program can decrease its effective complexity and thus increase its effective fitness by using ADFs.

Let C_{call} be the size or complexity of a call to an ADF, and C_{adf} the complexity of the overhead for an ADF definition. Let the original effective length of the complete program be C_{ej}, and the original absolute length be C_{aj}. A subfunction can then change the individual's effective fitness:

$$f_{ej} = f_j - p_c \cdot f_j \cdot \frac{C_{ej} - \Delta C_{ej}}{C_{aj} - \Delta C_{ej}} \cdot p_{dj} \qquad (7)$$

$$\Delta C_{ej} = \delta C_{block} \cdot (n-1) - C_{call} \cdot n - C_{adf2} \qquad (8)$$

As long as the change in effective fitness is positive, the individual will have an advantage in survival by using an ADF. Equation 7 motivates why a program sometimes can gain a reproduction advantage by spontaneously using modularization, and also explains why ADFs do not appear in simple problem spaces. Initial empirical investigations support this hypothesis, but a more rigid evaluation is planned in our future work.

2.4 Benefits from parsimony pressure

We have concluded, in the previous sections that a Genetic Programming system has an inherent tendency to promote solutions that have a short effective complexity. We have also seen that this could sometimes conflict with our goal of adapting to a specific fitness function.

It has previously been noted that a shorter overall length of an evolved program seems to results in a program with more generic behavior (Kinnear 1993, Tackett 1993). This could be made intuitively reasonable by many different examples. For instance, let us say that we want to perform symbolic regression of a function with the following fitness cases.
fn(3)=9, fn(0)=0, fn(2)=4, fn(1)=1
Two possible functions with maximal fitness' are:
First solution:
if x=0 then fn= 0 else, if x=1 then fn=1, else, if x=2 then fn= 4, else, if x=3 then fn=9, else, fn=0
Second solution:
fn=x*x
The second solution is shorter than the first and it behaves more uniformly for a larger set of input/output pairs, than the first solution. It could be argued that the second solution is more generic. This is the so called principle of Occam's Razor. In principle a solution has a greater probability of being general if it is shorter, provided that the functional and terminal set is not biased in an *unwanted way*. If the first solution in the example above was included as one of the

functions in the function set, then the function set could be regarded as biased in an unwanted way. The Principle of Occam's Razor can be formalized and put into a mathematical framework by algorithmic information theory and the Solomonoff-Levin distribution. For an excellent introduction to the relation between complexity and machine learning, see (Li 1990). Normally when we evolve an algorithm with a genetic programming system, or when optimizing parameters from data, we want to be able to apply the solution to a much wider set of inputs than the ones given by the fitness cases. We thus want a solution that is as generic as possible and - in analogy with above - we could say that we want a solution that is as short as possible or has the lowest complexity.

The pressure towards low effective complexity does not only work on the global program level, but also on the block level where short blocks have a higher probability of proliferation, c.f. the Schema Theorem. A genetic programming system can be regarded as employing a divide-and-conquer strategy towards the goal of finding a good solution with low complexity. This general problem solving strategy of "divide-and-conquer" with a continuous pressure toward elegant and generic sub-solutions could be one of the reasons for which a genetic programming system succeeds in reasonable time in such a broad set of domains.

The effects of this pressure should be balanced to avoid unwanted effects such as the inability to obtain a perfect fitness value, where one balancing factor could be an external pressure applied on the absolute size of the individual.

Notice that there is an important difference between a program with a short effective complexity and one with a short absolute complexity. The generalization properties of the program with short effective length could be decreased by introns that are not global introns. There might be blocks of code that only act as introns with the current fitness cases.
Example: (IF (< X 4) (X*X) (+ 0 0))
This program has a perfect score for the fitness cases above but will not give the desired result for input values above four. This example provides a motivation for applying external parsimony pressure, because it could remove the local intron. On the other hand, an external parsimony pressure could further increase the probability of unwanted effects. We propose to have a parsimony pressure that is balanced and variable during the evolution of the population. For instance, it could increase towards the end of the training session.

3 Empirical Results

In this section, we present a method for empirically measuring absolute complexity and effective complexity. We briefly present an example of these measurements and show how they support the hypothesis that there exists a compression pressure in the system.

The example is symbolic regression of a function using a polynomial with large constants. We have a set of ten fitness cases with input/output pairs taken from this polynomial function and we would like to evolve the function in the language of the register machine. For a more complete description of this experiment see (Nordin 1995b).

The register machine used performs arithmetic operations between a small set of registers. All instructions are coded as 32 bits. An instruction defines the destination registers, the two operands and the arithmetic operation to be used. One of the two operands can be a small constant, the other has to be another register. The operators used in this example are addition, subtraction and multiplication. All this information is stored in the 32 bits of the register. An individual consists of a continuous string of bits. Crossover is only allowed between the instructions at a locus that is a multiple of 32. The crossover operator selects two such points in two individuals and then swaps the instructions between them.

This approach enables us to cut and splice blocks of instructions into the individual without the risk of generating programs with invalid syntax. It also enables us to make a good estimation of the effective length of individuals. We do this by systematically replacing each instruction in an individual with a NOP (NoOperation) instruction that, by definition, has no effect on the state or output of the machine. If the individual still gives the same output for all fitness cases, then we know that the particular instruction substituted acted as an intron. The number of these instructions is added, and then subtracted from the absolute length to give a lower bound for the effective length of the program. This method gives a lower bound on the number of introns because it does not find higher order introns consisting of cooperating instructions such as the two instructions $a = a + 1, a = a - 1$. But higher order introns are themselves sensitive to crossover, and experiments show that the small proportion present in the first generations is rapidly substituted by the same number of first order introns. The estimation of effective length is thus close to the actual figure. In addition to crossover a mutation operator toggles bits in the individual with a certain probability. The selection operator in the examples below is fitness proportional selection. We have tried different selection operators resulting in different strength and dynamics of the system but with the same general results. The average standardized fitness during evolution of the function, plotted in Figures 2 and 3, shows the evolution of absolute length and effective length in the same experiment.

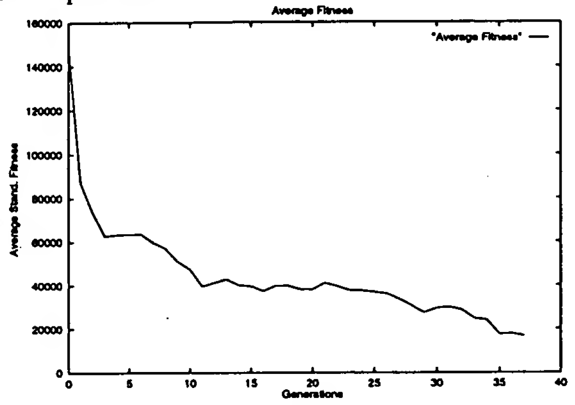

Figure 2: Average fitness, population size 2000.

Evolution of lengths starts from a point defined by the average initialization length. From generation 2 to 7, the rapidly decreasing fitness is the dominating term in the expression for effective fitness and the change in complexity is not dramatic. When the change in fitness becomes less important, the compression pressure increases and the effective lengths decrease. Finally, the absolute length starts to grow exponentially. Note that this happens while the effective length remains small and the average fitness continues to improve. Evolution over longer time has shown that the absolute complexity continues to grow exponentially without limit.

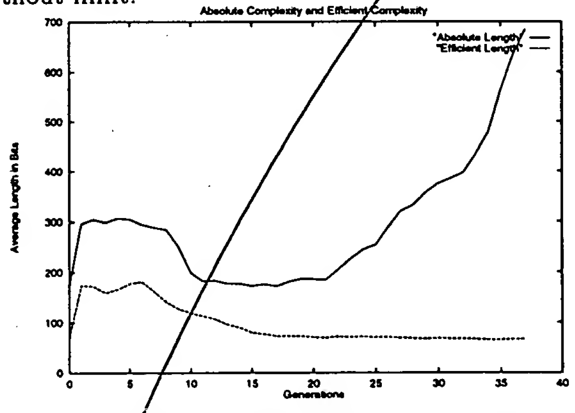

Figure 3: The evolution of the absolute and effective length.

To support the hypothesis that compression achieves its goal of protecting the individual, we have plotted the effect of crossover in different generations. Figure 4 shows the change of effects of crossover during evolution. This diagram consists of many diagrams of the same type as Figure 1 placed in sequence after each other. We can see that the absolutely dominating effects of cross-over are that either nothing happens to the fitness, or the fitness is worsened by more than 100 percent. The peak over the zero line increases which indicates a growingly unaffected fitness. The accumulated destructive effect of crossover to the left decreases after generation 15 as the ratio between absolute and effective length increases and the individual becomes more and more protected.

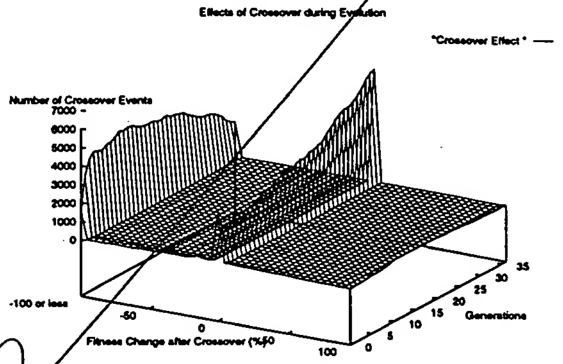

Figure 4: Distribution of crossover effect during evolution.

Figure 5 shows how a moderate constant external parsimony pressure can remove the introns completely after some generations. In this case, however, the negative effect of parsimony pressure is too strong and the system does not converge to an optimal solution. A smaller parsimony pressure would stop the exponential growth of the absolute length and force the two curves to follow each other more closely.

A tightly set maximum individual size can also be seen to increase the compression pressure, because the only way to increase the crossover target area when the maximum length is reached is to decrease the effective length. This manipulation sometimes leads to faster convergence but often the pressure becomes too high and the system fails to converge to an optimal solution.

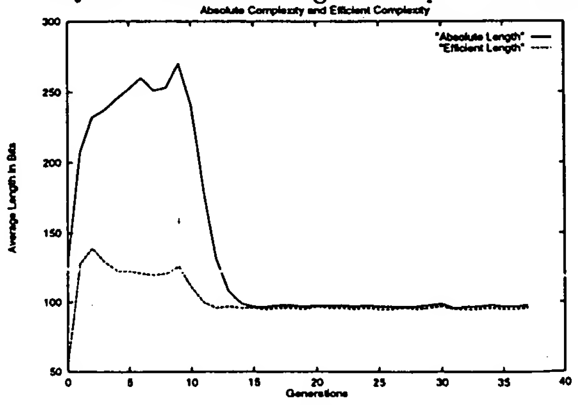

Figure 5: Effects of parsimony pressure.

4 Summary and Conclusions

We have argued for the existence of a compression pressure in evolutionary algorithms with variable length individuals. This pressure promotes short solutions with low complexity if the complexity is measured as effective complexity with introns removed. This phenomenon has both positive and negative effects, and is supported by data from evolution of computer programs. The positive effects are a more efficient search and more general behavior of the solution when used with unseen data. The negative effect is premature convergence towards a non-optimal solution. The key to the positive effects is to balance the complexity pressure according to the equation for effective fitness. This analysis should take into account a number of relevant properties, for instance representation, genetic operators, composition of fitness functions, crossover probability, generality of solutions, maximum depth or length of solutions, explicit parsimony, granularity of fitness function, initialization length and modularization.

Acknowledgement

The authors would like to thank Thomas Bäck and Sami Khuri for valuable comments to this paper. This research has been supported by the Ministry for Wissenschaft und Forschung (MWF) of Nordrhein-Westfalen, under grant I-A-4-6037.I .

References

Wolff, J.G. (1993) Computing, cognition and information compression. *AI Communications* 6(2):pp 107-127

J.P. Nordin,W. Banzhaf (1995) Evolving Turing Complete Programs for a Register Machine with Self-Modifying Code. In *Proceedings of the Sixth International Conference on Genetic Algortihms*,San Mateo, CA: Morgan Kaufmann Publishers

J. Levenick (1991) Inserting introns improves genetic algortithm success rate: Taking a cue from biology. In *Proceedings of the Fourth International Conference on Genetic Algortihms*, R.K. Belew and L.B. Booker (eds.) San Mateo, CA: Morgan Kaufmann Publishers Inc., pp 123-127

S. Forrest,M. Mitchell (1992) Relative building block fitness and the building block hypothesis In *Foundations of Genetic Algorithms 2*, D. Whitley (ed.). San Mateo, CA: Morgan Kaufmann Publishers Inc., pp 109-126.

J.D Watson,N.H. Hopkins,J.W Roberts,A.M Wiener (1987) *Molecular Biology of the Gene*, Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc.

J. Koza (1994) *Genetic Programming II*, Cambridge, MA: MIT Press.

J. Koza (1992) *Genetic Programming*, Cambridge, MA: MIT Press.

L. Altenberg (1994) The Evolution of Evolvability in Genetic Programming. In *Advances in Genetic Programming*, K. Kinnear, Jr. (ed.), Cambridge, MA: MIT Press. pp47-74.

J. Holland (1975) *Adaption in Natural and Artificial Systems*, Ann Arbor, MI: The University of Michigan Press.

P.J. Angeline, J.B Pollack (1993) Evolutionary Module Acquisition, In *Proceedings of the Second Annual Conference On Evolutionary Programming*, La Jolla, CA: Evolutionary Programming Society.

K. Kinnear (1993). Generality and Difficulty in Genetic Programming: Evolving a Sort. In *Proceeding of the fifth International Conference on Genetic Algorithms*, San Mateo, CA, Morgan Kaufmann.

W.A. Tackett (1993). Genetic Programming for Feature Discovery and Image Discrimination. In *Proceeding of the fifth International Conference on Genetic Algorithms*, San Mateo, CA, Morgan Kaufmann.

M. Li, P. Vitani (1990) Inductive Reasoning and Kolmorogov Complexity. In *Journal of Computer and System Sciences*, pp343-384

J.P. Nordin,F. Francone W. Banzhaf (1995b) Explicitly Defined Introns in Genetic Programming. Submitted to the GP workshop at *Machine Learning 1995*, Tahoe City, CA.

Genetic Programming Controlling a Miniature Robot

Peter Nordin[*] Wolfgang Banzhaf[†]
Fachbereich Informatik
Universität Dortmund
44221 Dortmund, Germany

Abstract

We have evaluated the use of Genetic Programming to directly control a miniature robot. The goal of the GP-system was to evolve real-time obstacle avoiding behaviour from sensorial data. The evolved programs are used in a sense-think-act context. We employed a novel technique to enable real time learning with a real robot. The technique uses a probabilistic sampling of the environment where each individual is tested on a new real-time fitness case in a tournament selection procedure. The fitness has a pain and a pleasure part. The negative part of fitness, the pain, is simply the sum of the proximity sensor values. In order to keep the robot from standing still or gyrating, it has a pleasure componentton fitness. It gets pleasure from going straight and fast. The evolved algorithm shows robust performance even if the robot is lifted and placed in a completely different environment or if obstacles are moved around.

1 Introduction

We have evaluated the use of Genetic Programming to control a miniature robot. To use a genetic process as the architecture for mental activities could, at first, be considered awkward. As far as we know today, genetic information processing is not directly involved in information processing in brains, but the idea of genetics as a model of mental processes is not new. Just 15 years after Darwin published *The origin of Species*, in 1874, the American psychologist William James argued that mental processes operate in a Darwinian manner. He suggested that ideas might somehow "compete" with one another in the brain leaving only the best or fittest. Just as Darwinian evolution shaped a better brain in a couple of million years, a similar Darwinian process operating within the brain might shape intelligent solutions to problems on the time scale of thought and action. This allows "our thoughts to die instead of ourselves". More recently, selectionist approaches to learning have been studied in detail by Gerald Edelman and his collaborators (see [1] and refernces therein).
The use of an evolutionary method to evolve controller architectures has been reported previously in a number of variants. Robotic controllers have, for instance, been evolved using dynamic recurrent neural nets [2], [3]. Several experiments have also been performed were a controller program has been evolved directly through genetic programming [4], [5], [6].

[*] email:nordin@ls11.informatik.uni-dortmund.de
[†] email: banzhaf@ls11.informatik.uni-dortmund.de

82

Previous experiments, however, with genetic programming and robotic control have been performed with a simulated robot and a simulated environment. In such a set-up, the environment and the robot can be easily reset to an initial state in order to ensure that each individual in the population is judged starting from the same state. Apart from being practically infeasible for a real robot, this method could result in over-specialization and failure to evolve a behaviour that can generalize to unseen environments and tasks. To overcome this last problem noise is sometimes artificially added to the simulated environment.
In our experiments we use a real robot trained in real time with actual sensors. In such an environment, the system has to evolve robust controllers because noise is present everywhere and the number of real-life training situations is infinite. In addition, it is highly impractical to reset the robot to a predefined state before evaluating a fitness case. Consequently, we had to devise a new method which ensures learning of behaviour while the environment is probabilistically sampled with new real-time fitness cases for each individual evaluation.

2 The Khepera Robot

Our experiments were performed with a standard autonomous miniature robot, the Swiss mobile robot platform Khepera. It is equipped with eight infrared proximity sensors. The mobile robot has a circular shape, a diameter of 6 cm and a height of 5 cm. It possesses two motors and on-board power supply. The motors can be independently controlled by a PID controller. The eight infrared sensors are distributed around the robot in a circular pattern. They emit infrared light, receive the reflected light and measure distances in a short range: 2-5 cm. The robot is also equipped with a Motorola 68331 micro-controller which can be connected to a SUN workstation via serial cable.
It is possible to control the robot in two ways. ~~The~~ controlling algorithm could be run on the workstation, with data and commands communicated through the serial line. Alternatively, the controlling algorithm is cross-compiled on the workstation and down-loaded to the robot which then runs the complete system in a stand-alone fashion. At present, the GP-system is run on the workstation, but we plan to port it and down-load it to the robot.
The micro controller has 256 KB of RAM and a large ROM containing a small operating system. The operating system has simple multi-tasking capabilities and manages the communication with the host computer.
The robot has several extension ports where peripherals such as grippers and TV cameras might be attached.

3 Objectives

The goal of the controlling GP system is to evolve obstacle avoiding behaviour in a sense-think-act context. The system operates real-time and aims at obstacle avoiding behaviour from real noisy sensorial data. For a more general description, definition and discussion of the problem domain see, [7], [8], [9], [10].
The controlling algorithm has a small population size, typically less than 50 individuals. The individuals use six values from the sensors as input and produce two output values that are transmitted to the robot as motor speeds. Each individual program does this manipulation independent of the others and thus stands for an individual behaviour of the robot if invoked to control the motors. The resulting

variety of behaviours does not have to be generated artificially, e.g. for explorative behaviour, it is always there, since a population of those individuals is processed by the GP system.

3.1 Training Environment

The robot was trained in two different environments. The first environment was a simple rectangular box. The dimensions of the box were 30 cm × 40 cm. The surface transmitted high friction to the wheels of the robot. This caused problems in the initial stages of training. Before the robot learned a good strategy it kept banging into the walls trying to "go through the walls". The high friction with the surface consequently stressed the motors.
We also designed a second, more complex environment. This larger environment is about 70 cm × 90 cm. It has an irregular boarder with different angles and four deceptive dead-ends in each corner. In the larger open area in the middle, loose obstacles can be placed. The friction between wheels and surface is considerably lower, enabling the robot to slip with its wheels during a collision with an obstacle. There is an increase in friction with the walls making it hard for the circular robot to turn while in contact with a wall.

4 The Evolutionary Algorithm

The GP-system is a steady state tournament selection algorithm [5], [11] with the following execution cycle:

1. Select k members for tournament.
2. For all members in tournament do:
   (a) Read out proximity sensors and feed the values to one individual in the tournament.
   (b) Execute the individual and store the resulting robot motor speeds.
   (c) Send motor speeds to the robot.
   (d) Sleep for 400ms to await the results of the action.
   (e) Read the proximity sensors again and compute fitness, see below.
3. Perform tournament selection.
4. Do mutation and crossover.
5. Goto step 1.

4.1 Fitness calculation

The fitness has a pain and a pleasure part. The negative contribution to fitness, called pain, is simply the sum of all proximity sensor values. The closer the robot's sensors are to an object, the more pain. In order to keep the robot from standing still or gyrating, it has a positive contribution to fitness, called pleasure, as well. It receives pleasure from going straight and fast. Both motor speed values minus the absolute value of their difference is thus added to the fitness.
Let p_i be the values of the proximity sensors ranging from $0 - 1023$ where a higher value means closer to an object. Let m_1 and m_2 be the left and right motor speeds

Table :	
Objective :	Obstacle avoiding behaviour in real-time
Terminal set :	Integers in the range 0-8192
Function set :	ADD, SUB, MUL, SHL, SHR, XOR, OR, AND
Raw and standardized fitness :	Pleasure subtracted from pain value desired value
Wrapper :	None
Parameters :	
Maximum population size :	30
Crossover Prob :	90%
Mutation Prob :	5%
Selection :	Tournament Selection
Termination criteria :	None
Maximum number of generations:	None
Maximum number of nodes:	256 (1024)

Table 1: Summary of parameters used during training.

resulting from an execution of an individual. The fitness value can then be expressed more formally as:

$$f = \sum p_i + \mid 15 - m_1 \mid + \mid 15 - m_2 \mid + \mid m_1 - m_2 \mid \qquad (1)$$

4.2 Implementation

The Evolutionary Algorithm we use in this paper is an advanced version of the CGPS described in [12], composed of variable length strings of 32 bit instructions for a register machine. The system has a linear genome. Each node in the genome is an instruction for a register machine. The register machine performs arithmetic operations on a small set of registers. Each instruction might also include a small integer constant of maximal 13 bits. The 32 bits in the instruction thus represents simple arithmetic operations such as "a=b+c" or "c=b*5". The actual format of the 32 bits corresponds to the machine code format of a SUN-4 [13], which enables the genetic operators to manipulate binary code directly. For a more thorough description of the system and its implementation, see [14].
The set-up is motivated by fast execution, low memory requirement and a linear genome which makes reasoning about information content less complex. This system is also used with the future micro-controller version in mind.
The system is a machine code manipulating GP system that uses two-point string crossover. A node is the atomic crossover unit in the GP structure. Crossover can occur on either or both sides of a node but not within a node. Because of our particular implementation of GP works with 32 bit machine code instructions , a node is a 32 bit instruction.
Mutation flips bits inside the 32-bit node. The mutation operator ensures that only the instructions in the function set and the defined ranges of registers and constants are the result of a mutation.
The function set used in these experiments are all low-level machine code instructions. There are the arithmetic operations ADD, SUB and MUL. The shift operations SLL and SLR and finally the logic operations AND, OR and XOR. All these instructions operate on 32-bit registers.
Table 1 gives a summary of the problem according to the conventions used in [4].

5 Results

Interestingly, the robot shows exploratory behaviour from the first moment. This is a result of the diversity in behaviour residing in the first generation of programs which has been generated randomly. Naturally, the behaviour is erratic at the outset of a run.
During the first minutes, the robot keeps colliding with different objects, but as time goes on the collisions become more and more infrequent. The first intelligent behaviour usually emerging is some kind of backing up after a collision. Then the robot gradually learns to steer away in an increasingly more sophisticated manner. After about 20 minutes, the robot has learned to avoid obstacles in the rectangular environment almost completely. It has learned to associate the values from the sensors with their respective location on the robot and to send correct motor commands. In this way the robot is able, for instance, to back out of a corner or turn away from an obstacle at its side. Tendencies toward adaption of a special path in order to avoid as many obstacles as possible can also be observed.
The robot's performance is similar in the second, more complex environment. The convergence time is slower. It takes about 40-60 minutes, or 200-300 generation equivalents, to evolve a good obstacle avoiding behaviour. The reason for the slower convergence time could be: less frequent collisions in the larger environment, the slippery surface, the high friction in collisions with walls and/or the less regular and more complex environment.

6 Future Work and Discussion

We have demonstrated that a GP system can be used to control an existing robot in a real-time environment with noisy input. The evolved algorithm shows robust performance even if the robot is lifted and placed in a completely different environment or if obstacles are moved around. We believe that the robust behaviour of the robot partly could be attributed to the built in generalisation capabilities of a genetic programming system [15].
Our next immediate goal is to cross-compile the GP-system and run it with the same set-up on the micro-controller on-board the robot. This would demonstrate the applicability of Genetic Programming to control tasks on low-end architectures. The technique could then potentially be applied to many one-chip control applications in, for instance, consumer electronics devices etc.
Another further extension to the system would be to eliminate the 400ms delay time of sleeping, during which the system is waiting for the result of its action. This could be achieved by allowing the system to memorize previous stimulus-response pairs and by enabling it to self-inspect memory later on in order to learn directly from past experiences without a need to wait for results of its actions. We anticipate the former to speed up the algorithm by a factor of at least 1000. The latter method would probably speed up the learning of behaviour by a comparably large factor.

Acknowledgement

One of us (P.N.) acknowledges support by a grant from the Ministerium für Wissenschaft und Forschung des Landes Nordrhein-Westfalen under contract I-A-4-6037-I

References

[1] Edelman G. (1987) *Neural Darwinism*, Basic Books, New York

[2] Cliff D. (1991) Computational Neuroethology: A Provisional Manifesto, in: *From Animals To Animats: Proceedings of the First International Conference on simulation of Adaptive Behaviour*, Meyer and Wilson (eds.), MIT Press, Cambridge, MA

[3] Harvey I., Husbands P. and Cliff D.(1993) Issues in evolutionary robotics, in: *From Animals To Animats 2: Proceedings of the Second International Conference on simulation of Adaptive Behaviour*, Meyer and Wilson (eds.), MIT Press, Cambridge, MA

[4] Koza, J. (1992) *Genetic Programming*, MIT Press, Cambridge, MA

[5] Reynolds C.W. (1994) Evolution of Obstacle Avoidance Behaviour, in: *Advances in Genetic Programming*, K. Kinnear, Jr. (ed.), MIT Press, Cambridge, MA

[6] Handley S. (1994) The automatic generation of Plans for a Mobile Robot via Genetic Programing with Automatically defined Functions, in: *Advances in Genetic Programming*, K. Kinnear, Jr. (ed.), MIT Press, Cambridge, MA

[7] Reynolds C.W. (1988) Not Bumping into Things, in: Notes for the *SIGGRAPH'88 course Developments in Physically-Based Modelling*, ACM-SIGGRAPH.

[8] Mataric M.J.(1993) Designing Emergent Behaviours: From Local Interactions to Collective Intelligence, in: *From Animals To Animats 2: Proceedings of the Second International Conference on simulation of Adaptive Behaviour*, Meyer and Wilson (eds.), MIT Press, Cambridge, MA

[9] Zapata R., Lepinay P., Novales C. and Deplanques P. (1993) Reactive Behaviours of Fast Mobile Robots in Unstructured Environments: Sensor-based Control and Neural Networks, in: *From Animals To Animats 2: Proceedings of the Second International Conference on simulation of Adaptive Behaviour*, Meyer and Wilson (eds.), MIT Press, Cambridge, MA

[10] Braitenberg V. (1984) *Vehicles*, MIT Press, Cambridge, MA.

[11] Syswerda G. (1991) A study of Reproduction in Generational Steady-State Genetic Algortihms, in: *Foundations of Genetic Algorithms*, Rawlings G.J.E. (ed.), Morgan Kaufmann, San Mateo, CA

[12] Nordin J.P. (1994) A Compiling Genetic Programming System that Directly Manipulates the Machine-Code, in: *Advances in Genetic Programming*, K. Kinnear, Jr. (ed.), MIT Press, Cambridge, MA

[13] The SPARC Architecture Manual,(1991), SPARC International Inc., Menlo Park, CA

[14] Nordin J.P. and Banzhaf W. (1995) Evolving Turing Complete Programs for a Register Machine with Self-Modifying Code, in: *Proceedings of Sixth International Conference of Genetic Algorithms, Pittsburgh, 1995*, L. Eshelman (ed.), Morgan Kaufmann, San Mateo, CA

[15] Nordin J.P. and Banzhaf W. (1995) Complexity Compression and Evolution, in *Proceedings of Sixth International Conference of Genetic Algorithms, Pittsburgh, 1995*, L. Eshelman (ed.), Morgan Kaufmann, San Mateo, CA

Programmatic Compression of Images and Sound

Peter Nordin and Wolfgang Banzhaf
Universität Dortmund LS11, Fachbereich Informatik, Dortmund, Germany
nordin,banzhaf@ls11.informatik.uni-dortmund.de

Abstract

The importance of digital data compression in the future media arena cannot be overestimated. A novel approach to data compression is built on Genetic Programming. This technique has been referred to as "programmatic compression". In this paper we apply a variant of programmatic compression to the compression of bitmap images and sampled digital sound. The work presented here constitutes the first successful result of genetic programming applied to compression of real full size images and sound. A compiling genetic programming system is used for efficiency reasons. Different related issues are discussed, such as the handling of very large fitness case sets.

1 Introduction

Programmatic compression is a very general form of compression. The basic idea behind this technique is that any system, which evolves programs or algorithms for generating data, can be viewed as a data compression system. The data that should be compressed are presented to the Genetic Programming system as fitness cases for symbolic regression. After choosing a function set that facilitates an accurate reproduction of the uncompressed data, the system then tries to evolve an individual program that, to a certain degree of precision, outputs the uncompressed data. If the evolved program solution can be expressed by fewer bits than the target data, then we have achieved a compression. In principle, this technique could address any compression problem with an appropriate choice of the function set and the method has a potential for both lossy and lossless compression. The initial studies performed by Koza [1]in the domain of very small bitmap images displays the difficulty of the problem and the huge computational effort needed for real world examples. We have applied a Compiling Genetic Programming System (CGPS) to the programmatic compression problem. A CGPS is up to 2000 times faster than a LISP Genetic Programming System (GPS) [5],[6], which enables the application of programmatic compression to real-world problems.

1.1 Compiling Genetic Programming

A CGPS is a GP system where all genetic manipulation of individuals is performed in binary machine code. Almost all computers today are of Von Neumann type, which means that programs and data reside in the same memory. Program information can thus be regarded as just another sort of data structure and it can be manipulated as any other data, in this case by the genetic operators. This arrangement enables the deletion of all interpreting steps, which results in a speed up of up to 2000 times compared to a LISP system, and 100 times compared to an interpreting C implementation. Some experiments described below consumed 10 CPU-days of execution. This kind of experiments would not be feasible with other types of GPSs. A LISP version of such an experiment could take more than 50 years while an interpreting C version still would need two years of execution time. These compression examples thus are real-life experiments and research, only feasible with a CGPS. For a more detailed description of the CGPS and comparison with other implementation methods, see [5],[6],[8].

2 Programmatic Compression (PC)

Programmatic image compression is briefly mentioned in [1]. Koza uses the 900 pixel values of a small 30x30 bitmap of a regular mathematical pattern as fitness cases for symbolic regression by a GPS. The system finds a good solution in less than 100 generations with a generation size of 2000 individuals and a function set consisting of arithmetic operators. This method has an important relation to the theoretically maximal compression achievable. If a Turing-complete function set is used then the shortest program that accurately produces the target data could be seen as one estimation of the Kolmogorov Complexity [3] which opens possibilities of a deeper theoretical analysis of the application. The GPS has at least the potential of finding this shortest possible compression of the data.

In our work we have used two real-world applications which both call for the use of very large fitness case sets. In the experiments described below we apply programmatic compression to bitmap images and sampled sound. The large potential of these two fields in the growing digital media arena cannot be overe[illegible]ted.

We examine the compression in several different ways for instance with different function sets, fitness evaluation methods and parameter settings.

The majority of experiments were performed with a simple function set consisting of the most usual low level machine code operations such as arithmetic and logic operations. It is surprising that the most efficient search runs were performed using these machine instructions as the function set, regardless of application. These results strongly support the universal applicability of GP. Sometimes critique has been heard that GP only would manage to solve different problems because there was a specially adapted function set corresponding to the problem domain. In most of our runs we have used a function set that is not decided by us but by the manufacturer of a certain computer. The same set of operators was used for the completely different domains of sound and images, suggesting that it is *not* the choice of function set that is primarily responsible for the success of GP.

The advantage of programmatic compression is the large flexibility possible with a programming language as output of a compression process. This flexibility implies a possible larger compression factor sometimes approaching the theoretical limit. The use of machine code instructions makes decompression very fast. The decompression algorithm is simply a small and efficient binary machine code program. Decompression rates of 150 million bits/second per processor are sometimes possible on a SUN 20. This figure definitely exceeds the demand of, for instance, real-time full size video.

Another advantageous feature of the system is that each element of the target data (sample or pixel) can be accessed independently of the values of other elements. For instance, a sample with a certain index number can be obtained without the need to calculate the values of samples before and after that index number.

2.1 Chunking

The simplest and most obvious method to be applied when using PC is to present the target data as a continuous sequence of fitness cases. The evolved program then tries to reproduce the entire list of data when executed. While uncomplicated and elegant, this method has some disadvantages when used with very large fitness case sets. If the target data are very complex, there is a risk that an evolutionary search will not converge to a solution with acceptable quality. It is also harder to predict how many generations are needed before a solution is found. Therefore, we have used two different systems applied to each of the two domains sound and images. The first system treats all the fitness cases at the same time. The other system applies programmatic compression to equally sized sub-sets of the fitness cases and evolves a solution to each of them. Below we refer to this method as *chunkin*[illegible]ime limit is imposed on each of the chunks which kee[illegible]e overall conversion time under control. The disadvantage of chunking is that the compression ratio could be lower since similarities between chunks will not be expressed in the individuals.

Chunking for sound is done by letting a CGPS evolve a solution for a fixed size sample chunk typically below the resolution time of the ear (70ms). In the image domain the pictures were divided into small quadratic blocks, 8x8 or 16x16 pixels each.

The system was in this way presented with fitness case sets of sizes from 32 to 100000 integers. The largest fitness case set corresponded to 10 seconds of sound or a 256x256x8bit pixel image.

2.1.1 Two methods for long term GP-memory

In order to speed up the search in the chunked data approach two different "memory methods" were introduced. The first method *kept* the entire generation between chunks which speeded up the search when similar chunks were adjacent. The second method made a few individuals "read-only" in every generation. They could take part in reproduction and crossover but only as parents. They were never allowed to be over-written by any offspring. For every generation, less than 10 individual were frozen, to help maintain diversity and to save valuable genetic code for future chunks. The second method has the advantage that genetic information can be reused between non-adjacent chunks. In this way, the GPS is given an association capability that enables it to use knowledge obtained from a similar problem earlier in time. The results of this experiments show that these mechanisms can save considerable amounts of search time while the best solutions found tend to converge to a slightly worse fitness. The tradeoff between search time and quality makes the method worth further investigation.

2.1.2 Multi-level Compression

When this method is used in practice it is advisable to compress the resulting machine code segments further. In most experiments only half of 32 bits in the instructions were used in the programs, the rest of the instructions did not have any relevance in a CGPS. This suggests that the instructions in the result can be further compressed by a factor of two. In addition the resulting "16-bit instructions" can be ordered according to their frequency and then Huffman-coded. This method will have only a marginal effect on the overall decompression speed, because decompression of programs only has to be performed once per chunk.
Note that introns will be removed before packing the programes. The introns will be identified with the method described in [2].

3 Compression of Soun

Our first experiments with PC concerns sound generating programs. The difficulty in generating and predicting large and complex time series has previous been reported by Oakley [4]. In this case, the time series consists of sampled sound. The data are produced by the built in sampler in a SUN workstation, which produces sound samples with 8 bit resolution and 8kHz sampling frequency. A recording of a one second sound, is thus 8000 bytes long and corresponds to 8000 fitness cases for the CGPS.

3.1 *Sound Generating Program architectures*

There are several program architectures possible for individuals for time series problems. The first obvious individual structure is a function without inputs that has the ability to store values as side-effects. The individual is placed in a loop, the variables initialized to zero and then output of the program only depends on what is stored by the program in these variables during the cycles of the loop. The variables can be accessed as a stack, as fixed variables or as index memory. In a similar way the individual can be directly fed by its previous output. A different approach is to feed the individual programs with the index number of the fitness case and expect it to compute the output only from this index number. A combination of side effects and index number is also possible. Other architectures of the individuals are also possible. The individual could also be fed with recent fitness case output, but most of these methods are not suitable for efficient compression, because you have to supply output values. This method, however can be used for time series prediction into the near future. Table 1 summarizes the parameters used during training.
Of these approaches the index version without side-effects gave the best results during evolution of sound. An integer is thus given as input to the individual program and the output is taken to be a single sound sample. The integer index is subsequently incremented and fed to the same individual, which then produces the next sample. In its simplest case, the fitness is just the sum of the absolute values of the differences between the actual sound samples and generated samples. There are many other methods to establish fitness values and as we have experienced, they can have a considerable impact on the quality of the evolved results.

3.2 *Fitness measurements*

Oakley has previously reported the importance of choosing an appropriate fitness measurement when evolving a chaotic and oscillating target function [4]. He observed that when fitness was measured as the difference

Terminal set :	Integers 0-8192
Function set :	ADD, SUB, MUL, SHL, SHR, XOR, OR, AND desired value
Maximum population size :	100 - 100000
Crossover Prob :	90%
Mutation Prob :	5%
Selection :	Tournament Selection
Maximum number of nodes:	16-1024

Table 1: Summary of parameters used during training.

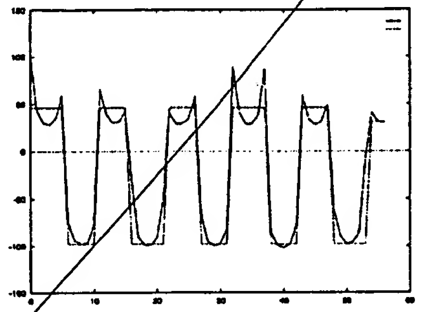

Figure 1: Adaption of generated sound to target sound (GP output is dotted)

between the target output and the actual output, the system would only converge to a constant output. The constant output represented the mean value of the target function. When we used the fitness measure of absolute value of differences, we noticed that for many initial generations, sometimes more than 100, the system displayed this behavior. However, a constant value is not of much use when the target is to evolve a sound. In the long run though the system always converged to a more accurate, oscillating output. Figure 3.2 shows the output of a program after the oscillation in the output has been adapted to a close fit of the target samples[1]. The original sound in this case is a sampled two-tone door bell. We can here see, how the output represented by the dotted line shows a very good adaption to the target function's frequency, amplitude and phase. A view of the entire sample also shows an adaption to the envelope, that is the long term changes in amplitude. However, the waveform in the output is simplified to a square wave. This behavior seemed hard to change by changing the parameters and varying the function set. When we changed the fitness function to the squared difference instead of the absolute value of the difference the result changed significantly. Not only did the convergence to an os-

[1]This figure, like most figures below, shows an excerpt of a longer sequence of samples. The details of the wave form of the full sequence is rather irregular, which might explain some irregularities in the output. It is impossible to show the full sequence in one diagram. If the training sequence was restricted to only the short sample spaces shown in the figures an even closer convergence would have been expected.

cillating output happen earlier, [illegible] resulting curve also displayed much more details. [illegible] also tried fitness differences raised to three and raised to four without significantly better results. The crucial point could be that non-linearity is needed.

The results with the squared error both looked and sounded much better, but the details of the waveform in the generated output lacked the extreme peaks of the target sample. To increase the selection pressure towards accurately reproducing these pointed features we tried with a difference between two adjacent points as the fitness value. This fitness measure did not work well when adopted alone but when it was in a weighted sum with the normal absolute value fitness it displayed a more pointed behavior.

All of the fitness methods above have the disadvantage that they do not take into account the way the ear perceives sound. The output can look quite similar to the samples and there could still be a lot of audible distortion. The evolved sounds often had metallic overtones and sometimes noise added to it. To some extent this could probably be filtered out with a filter similar to an anti-aliasing filter, but the fact remains that there is a difference between looking similar and sounding similar.

The human inner ear could be considered to perform a transformation similar to a Fourier transform of the sound we hear. This is the reason why we perceive tones and not quickly changing sequences of pressure variations. This mechanic Fourier transform is necessary because the 'clock frequency' of the brain is not fast enough to process sound directly.

Another fitness measurement would thus be a comparison with the Fourier transformed spectra of the generated samples and the target functions. Oakley [4] used Fourier transforms as a means to force the system away from local minima of constant output. Our system showed oscillation without the Fourier transform but we instead applied it to improve sound quality by aligning fitness function judgment of the output with that of the human ear.

We thus implemented a fast Fourier transform (FFT) taking real-valued data and transforming them into the frequency domain represented by complex numbers. Fitness is measured as the squared difference between the transform of output and the transform of the target samples. Both the real and imaginary part of the transform were separately taken into account by the fitness function in order to assure that the phase of the output was treated appropriately.

The sound produced by the FFT fitness system was audibly better. The CPU time needed for this system was about twice that of the system without FFT, which was reasonable considering the quality improvement.

When FFT is applied in the fitness function then the exact amplitude of the curve is less important, instead

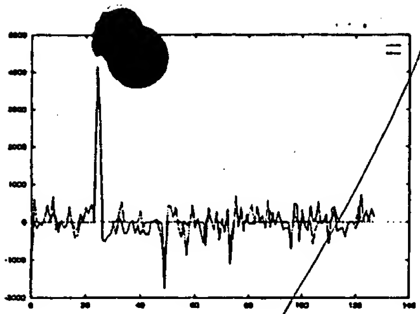
Figure 2: FFT spectrum of the sample(GP output is dotted)

features that influence the overtones of the system are dominating. Figure 3.2 shows the spectrum of the same sample. An external function in a CGPS is a function that is not part of the processor's instruction set and has to be used by a call instruction. Any C-function can be linked to the system and used as an external function.
The function set in the experiments presented above was eight low-level machine code instructions: addition, subtraction, multiplication, shift left, shift right, logical and, logical or and logical exclusive or.
Different experiments where also performed with the use of side effects in functions. The programs were given the possibility to store values in memory between the processing of the samples. In the extreme case the individual was not provided with an index number for the sample that should be generated instead the individual had to do its own book keeping with the use of memory variables. These programs took longer time to evolve and performed worse in fitness but had a softer sound with less overtones. Side effects were introduced as Save/Restore functions,the Swap function and indexed memory of different sizes.
The first reflection over sound generating programs is that it would be advantageous to use functions related to oscillation in the function set. We therefore tried to use the sinus function directly in the function set as well as functions that are components of a discrete differential equation for oscillation and thus together could produce a variety of different curves. To our surprise did none of these functions perform better than the individuals built with the simple machine code building blocks.
Automatically defined subfunctions (ADFs) [2] were also used without significantly improving results.

3.3 *Evolving Large Samples Sequences*

As mentioned above two methods were tried for evolution of large samples sequences. The simple linear method that tried to evolve the sample with one individual proved insufficient for large and complex sound. This method also had the disadvantage that its convergence behavior and evolution time requirements were

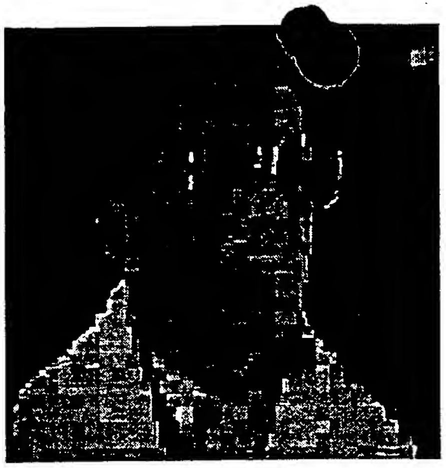
Figure 3: Target bitmap with fast evolution

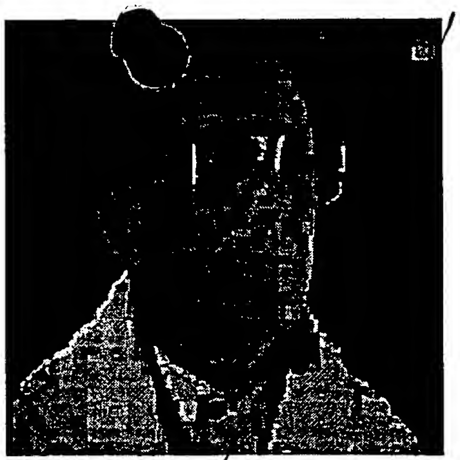
Figure 4: An example of output from the programmatic compression system

hard to predict. It was possible to evolve the frequency, amplitude, phase, envelop and, to some extent, the wave form of a 16.000 samples sound lasting two seconds, if the sound was regular. However, to evolve changes in frequency in a sample, turned out to be harder. We succeeded to generate two different tones, that change at a specific point in the sample frequency, but we did not have any good results with more complex sounds. The more random output from earlier generations sometimes expressed complex melodies, but it did not evolve such melodies corresponding to a specific fitness case set. Even more complex sounds, like spoken words were impossible to mimic with this linear method and with sample length of audible size.
The solution we use instead for evolution of longer and more complex sounds is chunking. The sample sequence is divided into equally sized sequences and the CGPS in then applied to each chunk in turn. The chunk size is chosen so that the length of a chunk is below the resolution of the human ear. Pulses that come faster than 70ms are perceived as tones and not individual pulses. The fact that the evolved sound always is in phase assures that the sound pieces fit together well in the joint of two chunks. With this method it is possible to evolve complex sound lasting for a longer time. We have for instance evolved human speech with an understandable quality. Melodies of 100000 samples have also been evolved. In future work we would like to investigate the possibility of evolving the division points between chunks allowing for adaptive and varying chunk lengths.

3.4 Results

Compression ratios vary much depending on the regularity of the target sound. For regular sound the compression ratio can easily be 10 times or more, while the ratio for speech is around two times. The quality is still below telephone quality but we will in the future investigate several methods to improve it.

The CPU time for generating a ten second chunked sample is in the range of tens of hours on a SUN 20.

4 Compression of Pictures

Our system for programmatic compression of images bears many similarities with the sound compressing system. Instead of trying to evolve a one dimensional array of samples we here try to evolve a two dimensional array of pixels. In our experiments every pixel holds 8 bits of information. In the experiments we have used gray scale images because of the technically more complex handling of color in the display palette. The technique is however equally applicable to color pictures and to pictures with more than 8 bit quantification, i.e. 24 bit color. The output of the evolved individuals consists of eight bit numbers. If very fast decompression is required, it is possible to use the full 32 integer bits of the architecture. In this way more bits per second can be decompressed.

4.1 Program architecture

Basically the same method is used for images and sound. As a first method an individual is fed by the X and Y coordinates of the pixel and the output is interpreted as the value of the pixel. However a simpler method proved to give better results. Instead of supplying both the X and Y coordinates to the programs, we supplied a *one dimensional* index value of the pixel.

Chunking is a must when working with realistically sized images. In our experiments we have used pictures consisting of 256x256 pixel which were divided into squares of either 16x16 or 8x8 pixel. The definitely best results were achieved with 8x8 pixel blocks. This size corresponds to the size used by several other image compression techniques such as JPEG [9]. Different methods for side effect were also used, and unlike the sound example the pictures had a slightly better quality when the

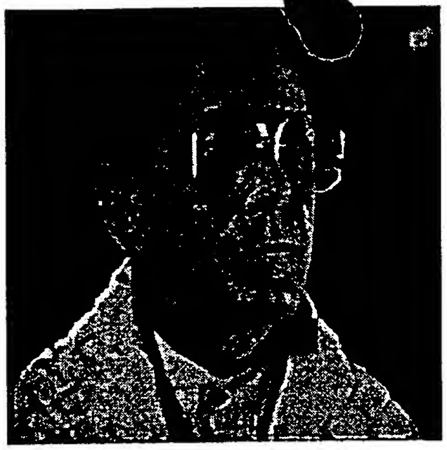

Figure 5: The target bitmap

programs also had access to a small set of memory cells.

Figure 4.1 shows an example of an evolved image that used figure 4.1 as target data. Figure 3.2 shows a quickly (one hour) evolved picture with lower quality .

4.2 Fitness functions

Different fitness measurements were also applied, and similar to the sound domain, the squared error differences gave better results than the absolute value.

In future work we would like to use two dimensional FFT in the fitness function, as well as functions measuring differences between adjacent pixels in several directions.

The blocks of chunking are visible to a varying degree in the decompressed pictures. By incorporating the differences in color around the edges of blocks into the fitness function we hope to reduce this effect. We will also try different conventional smoothing techniques used i.e. with JPEG for the same purpose.

The CPU times for compressing a picture in our experiments on a SUN20 are in the range of 30 minutes to 10 days depending on the requested quality and compression ratio.

5 Summary and Conclusion

We have described experiments that have shown that programmatic compression can be used with other than toy problems in both the image and sound domain.

With these compression experiments we have demonstrated that GP can be used with very large fitness case sets if a compiling approach is applied. Some of our experimental runs have evaluated hundreds of billions of fitness cases, something that would take years on an interpreting GP system. The machine code approach provides for very fast decompression suitable even for the demand of real time full size video.

Acknowledgm

One of us (P.N.) acknowledges support by a grant from the Ministerium für Wissenschaft und Forschung des Landes Nordrhein-Westfalen under contract I-A-4-6037-I

Bibliography

[1] Koza, J. (1992) *Genetic Programming*, Cambridge, MA: MIT Press.

[2] Koza, J. (1993) *Genetic Programming II, Automatic discovery of reusable programs*, Cambridge, MA: MIT Press.

[3] Li M., Vitani P. (1990) Inductive Reasoning and Kolmorogov Complexity. In *Journal of Computer and System Sciences*, pp343-384

[4] Oakley H. (1994) Two Scientific Applications of Genetic Programming: Stack Filters and Non-Linear Equation Fitting to Chaotic Data In *Advances in Genetic Programming*, K. Kinnear, Jr. (ed.), Cambridge, MA , USA, MIT Press.

[5] Nordin, J.P. (1994) A Compiling Genetic Programming System that Directly Manipulates the Machine-Code. In *Advances in Genetic Programming*, K. Kinnear, Jr. (ed.), Cambridge, MA , USA, MIT Press.

[6] Nordin, J.P. , Banzhaf, W. (1995) Evolving Turing Complete Programs for a Register Machine with Self-Modifying Code. In: *Proceedings of the Sixth International Conference of Genetic Algorithms*, Pittsburgh, Penn., USA, Morgan Kaufmann Publishers

[7] Nordin, J.P. , Banzhaf W. (1995) Complexity Compression and Evolution. In: *Proceedings of the Sixth International Conference of Genetic Algorithms*, Pittsburgh, Penn. ,USA, Morgan Kaufmann Publishers

[8] Nordin, J.P. (1995) Compiling Genetic Programming. To appear in: *Handbook of Evolutionary Computation*, T. Bäck, D. Fogel (ed.), New York, NY ,USA , Oxford University Press.

[9] Pennebaker, W.B., Mitchell, J.L. (1993) JPEG still image data compression standard. Van Nostrand Reinhold, New York , USA

Genetic Reasoning
Evolving Proofs with Genetic Search

Peter Nordin[*] Wolfgang Banzhaf[†]
Fachbereich Informatik
Universität Dortmund
44221 Dortmund, Germany

January 21, 1996

Abstract

Most automated reasoning systems relies on human knowledge or heuristics to guide the reasoning or search for proofs. We have evaluated the use of a powerful general search algorithm to search in the space of mathematical proofs. In our approach automated reasoning is seen as an instance of automated programming where the proof is seen as a program (of functions corresponding to rules of inference) that transforms a statement into an axiom. Genetic programming is a technique for automated programming that evolves programs with a genetic algorithm. We show that such a system can be used to evolve mathematical proofs in complex domains i.e. arithmetics and program verification. The system is not restricted to evaluations of classical two-valued logic but can be used with for instance Kleene's three valued logic in order to detect paradoxes that can occur in real life reasoning applications.

[*] email:nordin@ls11.informatik.uni-dortmund.de
[†] email: banzhaf@ls11.informatik.uni-dortmund.de

94

1 Introduction

We present an approach to reasoning that uses a genetic search heuristic to navigate and search in the space of true statements. An algorithm inspired by natural selection and survival of the fittest is used to search for proofs.
To use a genetic process as the architecture for mentally related activities could, at first, be considered awkward. As far as we know today, genetic information processing is not directly involved in information processing in brains, though the idea of genetics as a model of mental processes is not new. William James, the father of American psychology, argued just 15 years after Darwin published *The Origin of Species*, in 1874, that mental processes could operate in a Darwinian manner (William James 1890). He suggested that ideas "compete" with one another in the brain leaving only the best or fittest. Just as Darwinian evolution shaped a better brain in a couple of million years, a similar Darwinian process operating within the brain might shape intelligent solutions to problems on the time scale of thought and action. This allows "our thoughts to die instead of ourselves".
Evolutionary Algorithms mimic aspects of natural evolution, to optimize a solution towards a defined goal. Darwin's principle of natural selection and survival of the fittest, which is thought to be responsible for the evolution of all life forms on earth, has been employed successfully on computers over the past 30 years. Different research subfields have emerged such as , Evolution Strategies (Schwefel 1975,1995), Genetic Algorithms (Holland 1975) and Evolutionary Programming (Fogel, Owens & Walsh 1966), all mimicking various aspects of natural evolution. In recent years, these methods have been applied successfully to a spectrum of real-world and academic problem domains.
A mathematical function can for instance by optimized by a genetic algorithm that keeps a population of solution candidates which are reproduced by selection, modified by mutation and recombination during evolution until a sufficiently good solution is found.
A comparatively young research topic in this field is Genetic Programming (GP). Genetic Programming uses the mechanisms behind natural selection for *evolution of computer programs*. The search space is here the space of all computer programs. This contrasts other evolutionary algorithms which often optimizes real numbers or vectors of real numbers. In GP the structures optimized is a symbolic representation of a computer program. Instead of a human programmer programming the computer, the computer can modify, through genetic operators, a population of programs in order to finally generate a program that solves the defined problem. The GP technique, like other adaptive and learning techniques, has applications in problem domains where analytical solutions are incomplete and insufficient to the human programmer or when there isn't enough time or resources available to allow for human programming.
Methods related to GP were suggested as far back as in the 1950s (Friedberg 1958). For various reasons these experiments never were a complete success even

95

if partial results were achieved (Cramer 1985). A breakthrough came when Koza formulated his approach based on program individuals as tree structures represented by LISP S-expressions (Koza 1992). His hierarchical subtree exchanging genetic crossover operator guaranteed syntactic closure during evolution.
GP differs from other Evolutionary techniques and other "soft-computing" techniques in that it produces symbolic information (i.e. computer programs) as output. It can also efficiently process symbolic information as input. Despite this unique strength, has genetic programming mostly been applied in numerical or boolean problem domains.
In this paper we exploit GP's strength of processing purely symbolic information by searching in the domain of proofs.
Genetic Programming is thus a method for automated programming. A formal proof of a statement could be seen as a computer program and a theorem prover as an application of automated programming. The proof program is a list of inference functions transforming a statement to an axiom (or to a statement known to be false, i.e. a contradiction). Rules of inference are here seen as functions from theorems to theorems, like in for instance the programming language ML. The inference rules are rules that matches a part of a formula and rewrites it as something equivalent, or equally true. The formula $X + 0$ could for instance be replaced by X, as one of the axioms of Peano arithmetic tells us. This rule describes a function from theorem to theorem. In the same way the reverse is true and X could be replaced by $X + 0$, (but this is considered as an other function.)
This simplest form of theorem prover systematically applies rules of inference to construct all possible valid logical deductions. This was what the pioneering AI research tried in the 1950s. Most notably the Logic Theory Machine of Alan Newell and Herbert Simon (Newell and Simon 1956). In practise can such a method only find very short proofs. The combinatorial explosion will quickly exhaust any computer resources. Different more efficient variants of representation and search methods has been introduced like the *resolution* method pioneered by, for instance, Robinson in the early 1960s, (Robinson 1965), (Bundy 1983). These methods were more adapted to machine reasoning then human reasoning and were more efficient when implemented. Still they needed to by governed by strategies and heuristics optimizing the order in which clauses were resolved etc. Resolution theorem provers help against the combinatorial explosion but they do not eliminate it. They can still only produce proofs of modest length.
The disappointment of some of the reasoning system lead to the conclusion that more human knowledge needs to be put into the reasoning process, or as Bledsoe put it (Bledsoe 1977):

> The word "knowledge" is a key to much of this modern theorem-proving. Somehow we want to use the knowledge accumulated by humans over the last few thousand years, to help direct the search for proofs

This knowledge is included as heuristics, weights and priorities in the theorem prover. If it is an *interactive theorem prover* it can have its heuristics modified by a human during execution. Regarding search algorithm most systems rely on a hill-climbing algorithm, back-tracking or a best-first heuristic (Winker and Wos 1978).
In our research we are investigating another approach. Instead of using explicitly added heuristics to guide the search we apply a more powerful and robust general search algorithm. The hypothesis is that the robustness of genetic search could free the reasoning system from some of the burdens of carrying specialized heuristics. The search could then be more autonomous and act more "intelligent" when it produces solution to problems with less a priori knowledge.

2 Genetic Programming

Genetic programming (Koza 1992) uses an evolutionary technique to *breed* programs. First a goal in the form of a goodness criteria is defined. This, so called, fitness function could for instance be the error in a symbolic regression function. The population – a set of solution candidates – is initialized with random contents, (random programs). Each "generation" the most fit individual programs are selected for reproduction. These highly fit individuals have offspring trough recombination (crossover) and mutation. Various methods exists for selection and reproduction but the idea is that the better individuals, and their offspring, gradually replace the worse performing individuals [1].
The individual solution candidate is represented as a tree (the genome). This tree can be seen as the parse tree of the program in a programming language. Recombination is normally performed by two parents which exchange subtrees ,see figure 1.

A typical application of GP is symbolic regression. Symbolic regression is the procedure of inducing a symbolic equation, function or program which fits given numerical data. Genetic programming is ideal for symbolic regression and most GP applications could be reformulated as a variant of symbolic regression. A GP system performing symbolic regression takes a number of numerical input/output relations, called fitness cases, and produces a function or program that is consistent with these fitness cases. Consider for example the following fitness cases:

```
f(2) = 6
f(4) = 20
f(5) = 30
f(7) = 56
```

[1] Genetic programming has some similarities with, "beam search" (Lowerre and Reddy 1980), (Rosenbloom 1987), if the population is regarded as the memory buffer and the fitness as a stochastically assigned priority.

97

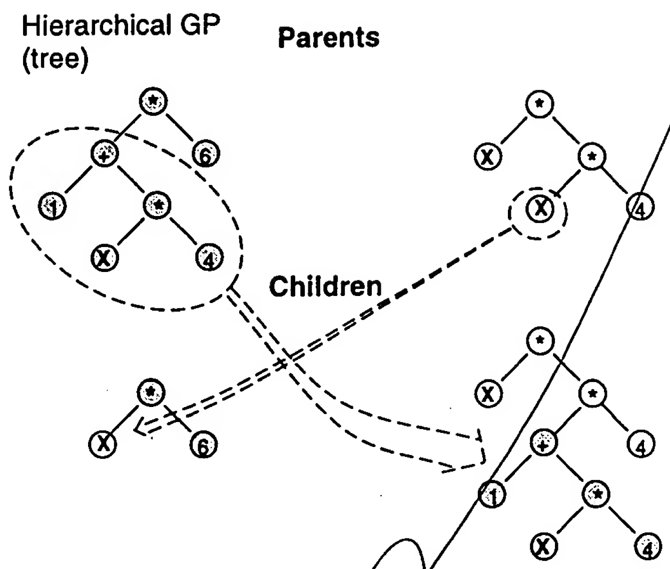

Figure 1: hierarchical crossover in Genetic programming.

One of the infinite number of perfect solutions would be $f(x) = x^2 + x$. The fitness function would, for instance, be the sum of difference between an individual's (function's) actual output and the output specified by the fitness cases. The *function set*, or the function primitives could in this case be the arithmetic primitives $+, -, \cdot, /$, as also seen in figure 1.
The two most important decisions to make before training a genetic programming system is to choose a good fitness function and to choose the right *function set*.
The fitness function should allow for a gradual improvement during evolution and it should, like other objective functions, give meaningful feedback to the GP induction system.
The function set should contain relevant primitives to the problem domain. Each function in the function set should also be syntactically closed – they should be able to gracefully accept all possible inputs in the problem domain.

3 Genetic Reasoning

In order to apply GP to reasoning and automated theorem proving (ATP) we need to design the appropriate fitness function, function set and choose a theorem representation. Our goal is to handle statements about arithmetics in a

logic as powerful as first order logic.
The function set is made up of function representing rules of inference. Such a function could for instance be the rule $X+O$ *can be replaced by* X. All function we use are unary-functions – they take one statement as input and produce another equivalent statement. This means that the tree representation of the individuals in GP collapses into a linear list representation[2] and that recombination will exchange linear segments of the individual genome, see figure 2.

The actual statement that should be proven true or false is represented by a

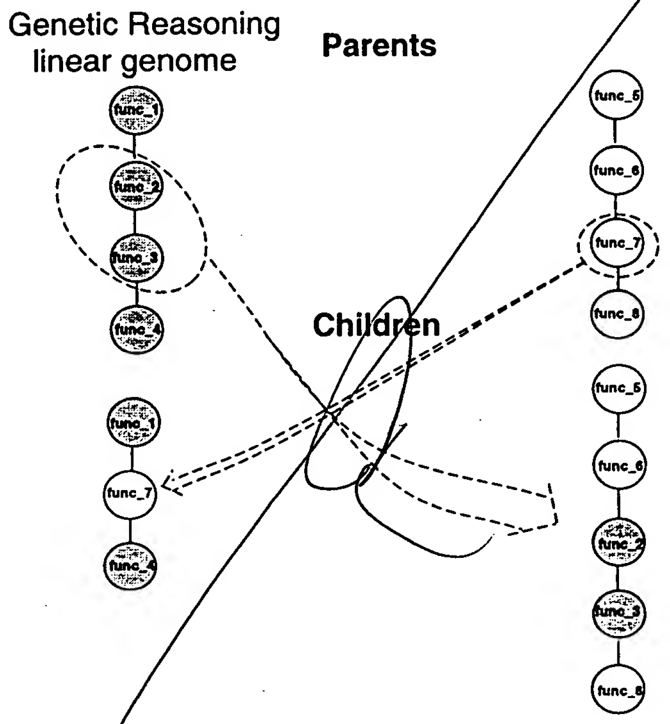

Figure 2: Crossover in Genetic Reasoning.

tree. Universal quantification is indicated by leaving variables free. Existential quantification is represented by a Skolem function, as common in several approaches to ATP. The natural numbers are built into our system in the form of the *zero (0)* symbol and the successor function. Figure 3 shows how the (false) statement $3 = 2 + 0$ would be represented.

The inference functions in the genome are then applied in turn to this structure.

[2]Note that in this application the genome structure is linear while the fitness case input is a tree structure. This is sometimes the other way around in other GP applications.

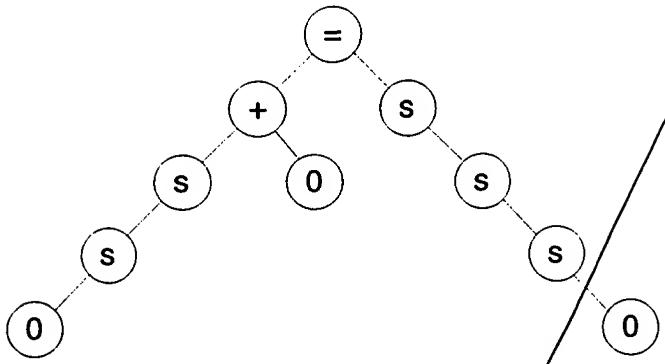

Figure 3: Representation of the statement $3 = 2 + 0$.

When all inference function have been applied – when the individual "program" has terminated – then we have another tree structure representing an equivalent statement. Let us say that we call the rule $X + O$ can be replaced by X, $func_1$. If this function is part of the genome it will try to match a subtree in the statement and, if it finds a match, replace it with X, see figure 4.

In figure 4 the function matches a sub-tree in the statement and the statement

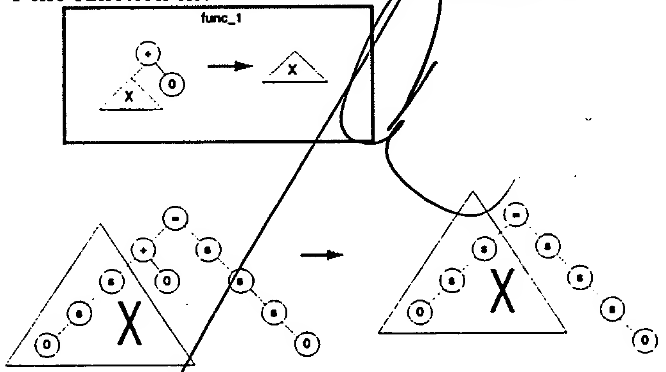

Figure 4: Application of $func_1$ to the statement $3 = 2 + 0$.

can be transformed. With this transformation the size of the statement structure is reduced, but there exists an equal number of functions in the function set that increase the size of the structure. If a function *does not* match any sub-tree in the statement then the structure is left untouched. This procedure provides syntactic closure and does also give the opportunity to temporary store

100

unused material in the genome. The phenomena of unused genetic material is called introns in biology and may play an important role in the efficiency of a genetic search, see (Nordin, Francone and Banzhaf 1995b), (Nordin and Banzhaf 1995a).

3.1 The Fitness Function

The fitness function is very simple in our genetic reasoning system. It is just the number of nodes in the statement structure (figure 3). The two simplest and shortest statements are the Boolean constants **t** and **f**, each represented by only one node. These truth values are short hand for an axiom respectively a contradiction. The genetic system will thus try to simplify any expression down to the statement of either true or false, represented by the nodes **t** and **f**.
Genetic search has been proven to perform well and robustly in a wide variety of highly multi-modal search domains were local optima easily can trap a more hill climbing related approach. So, the pressure towards simplifying the statement does not mean that the system will try to constantly shorten the structure. The concept of a population of solution candidates helps the search to avoid local optima. The selection criteria from generation to generation does not monotonically select the best individuals but probabilistically reproduces individual with a large variation in fitness.

4 The Logic of Genetic Reasoning

The logic of the genetic reasoning system is similar to the one of the automated reasoning system Nqthm (Boyer and More 1979). It is a quantifier free first order logic with equality. The rules of inference are from propositional logic and the equality. Mathematical induction is an important part of the system. This principle is added explicitly as it cannot be expressed efficiently in first order logic.
Functions defining all boolean arithmetic operations are built-in ($\wedge, \vee, \neg, \rightarrow$).
The boolean constants **t** and **f** representing an axiom respectively a contradiction are also built-in. There are if-then-else functions as well as equality.
The natural numbers and arithmetics are defined by the peano axioms and the symmetry relation.
The deduction theorem is also predefined.
It is possible to add functions defining abstract data types and lemmas to support a specific application. In the register machine example below axioms describing this fictional processor are added.

4.1 The Evolutionary Algorithm and its Implementation

It is in principle possible to use any variant of GP as the basis for genetic reasoning. We have used a steady-state algorithm with tournament selection. The size of the population has been between 100 – 1000 individuals.
The GP system is implemented on a SUN-20 in PROLOG. The built-in features of PROLOG, such as pattern matching and list handling, simplifies implementation significantly

5 Results

Our reasoning system has so far been applied to two different domains: proving simple statement in arithmetics and reasoning about, for instance, halting of machine code programs. Both these applications relies to a great extent on mathematical induction as the proof method. All examples are such that the proof could not be obtained by simplification only. The system could not just hill-climb towards a solution, instead various steps of expansions needed on the way, to be able to finally reach a constant false or true statement. These expansions where not defined by a lemma or heuristic but were the result of the genetic search process.

5.1 Arithmetic Problems

The arithmetic problems that we started our evaluation with were selected using two criteria. The statement should be hard to prove without induction and it should be impossible to prove by just transformations to shorter statements. The induction principle might in it self require proofs that cannot be obtained by monotonic transformations and reductions. A typical statement used is:

There is no natural number bigger than three, that when added two to it, is equal to four. This statement is represented by the genome (tree structure) in figure 5.

This kind of statement can be proven (false) with a few hundred generation equivalents and a population size of 200 individuals. This calculation takes about 10 minutes on our SPARC-20.

5.2 Termination Proofs of a Program for a Register Machine

A register machine is a machine that operates with instructions that manipulates a limited set of registers. All CPUs in commercial computers are register machines. The instructions of a register machine might looks as: $a = b + 12$ which should be interpreted as: add 12 to the content of register b and place the result in register a. The processor also contains instructions for control of program flow, for instance jumps as well as conditional instructions. The axioms

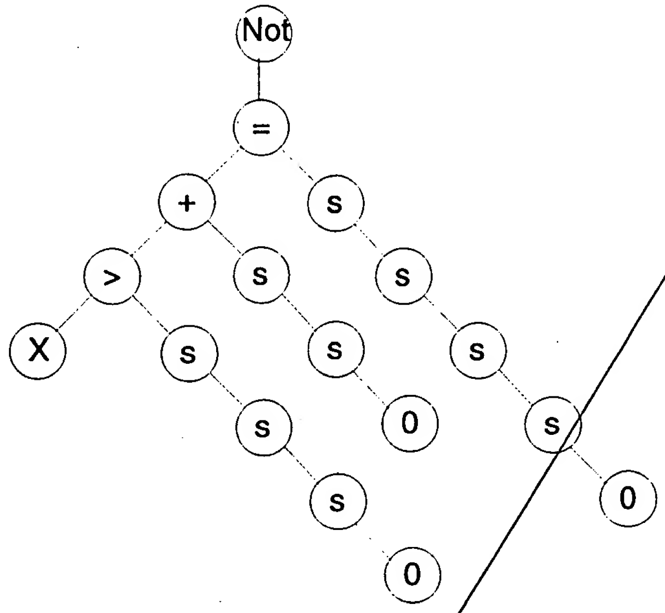

Figure 5: An example statement representation.

defining the processor and the current processor program is added to the system. We then use the genetic reasoning system to determine the correctness of the machine code programs, which often means the proof of termination. This approach demonstrates one of the strengths of the genetic reasoning system as termination proofs almost exclusively requires induction to be part of the system.

The machine code application is slightly more complex than the purely arithmetic statements and the verification of a short program, 2-10 instructions, takes about one hour with a population size of 1000 individuals.

The correctness proofs of programs has many applications, data security, high robustness in programs (i.e. satellite technology), simplification of machine code programs, (Boyer and Yu 1992). GP is often accused of producing non-robust solutions. A reasoning system could judge evolved solutions to prove if they are complete or not. Much like a human programmer that first almost by intuition might put a program solution together and when it works he (hopefully) studies it and reasons in his mind to see if it will really hold for all input and if it is safe, if it can be simplified etc. The termination proofs could also be used with a normal GP system to detect infinite loops in individual programs during evolution. Normally a few hundred generation equivalents has been needed to reach the true and false constants in our program verification experiments.

5.3 Multi-valued Logic

Other reasoning techniques, such as resolution, relies in a classical two valued logic. Most of these systems search for a contradiction to disprove a statement and to conclude its negation. Genetic reasoning on the other hand can be used with any logic. It is just a matter of defining the right transformation functions in the function set. We have implemented Kleene's three valued logic (Kleene 1950) in the system to better deal with paradoxes. In Kleene's tree valued logic a formula is either true false or a paradox (catch-22). With this logic our system can answer statement of paradoxical character. We have for instance included the definition of the genetic reasoning system as a primitive in the computer language we are reasoning about. This can give rise to true paradoxes that are hard to resolve in classic logic. We have also tried the use of a fourth truth value which is needed in the case when the genetic reasoning system does not find a proof of the statement. This fourth truth value represents a n "unknown" truth.

6 Future Work

In our experiments we have so far been concentrated on the problem of proving a theorem. We are, however, convinced that the genetic reasoning method has application is less rigid areas of reasoning ~~and~~ machine learning, such as planning in robotics. We plan to continue and extend our robot experiments on the Khepera robot platform with the application of genetic reasoning (Nordin and Banzhaf 1995c).
We also plan to port the system to C which will give an acceleration of as much as 100 times. This would allow us to try more difficult problems using large population sizes.

7 Summary and Conclusions

We have demonstrated that automated reasoning could be seen as an instance of automated programming. In this spirit we a have evaluated the use of a robust genetic search algorithm to search the spaces of proofs. The system has been able to avoid local minima in its search and found proofs of statements from complex domains such as arithmetics and program verification. The system uses no heuristic or human knowledge to guide its search, instead it relies on the performance of the search algorithm. We believe that this technique can have applications in many automated reasoning, and machine learning domains for instance robot planning.

Acknowledgement

Thanks to Steffo Weber for invaluable advise in the process of writing this paper. We would also like to thank Tom Haynes for giving us pointers to his experiments with GP and theorem proving (Knight and Haynes 1994). One of us (P.N.) acknowledges support by a grant from the Ministerium für Wissenschaft und Forschung des Landes Nordrhein-Westfalen under contract I-A-4-6037-I

References

[1] James W. (1890) The principles of psychology Vol.1. Originally published: Henry Holt, New York1890.

[2] Schwefel, H.-P. (1995) *Evolution and Optimum Seeking*, Wiley,New York

[3] Holland, J. (1975) *Adaption in Natural and Artificial Systems*, Ann Arbor, MI: The University of Michigan Press.

[4] Fogel, L.J., Owens, A.J., Walsh, M.J., (1966) Artificial Intelligence through Simulated Evolution. Wiley, New York

[5] Friedberg, R.M. (1958) A Learning Machine - Part I,*IBM Journal of Research and Development* IBM, USA 2(1), 2-11.

[6] Cramer, N.L. (1985). A representation for adaptive generation of simple sequential programs. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, pp183-187

[7] Koza, J. (1992) *Genetic Programming*, MIT Press, Cambridge, MA

[8] Newell, A., Shaw J.C., and Simon H. (1957) Empirical Explorations of the Logic Theorem Machine: A case study in Heuristic, in *Proceedings of Western Joint Computer Conference* Vol. 15.

[9] Robinson J.A., (1965) A Machine Oriented Logic Based on the Resolution Principle, In *J.ACM*, Vol. 12, No. 1, pp. 23-41.

[10] Bundy A., (1983) The Computer Modeling of Mathematical Reasoning, Academic Press, London, pp. 74-77.

[11] Bledsoe W. W., (1977) Non-Resolution Theorem Proving, In *Artificial Intelligence*, Vol. 9, pp 2-3.

[12] Winker S., Wos L.,, (1978) Automated Generation of Models and Counterexamples and its application to Open Questions in Ternary Boolean Algebra, In Proceedings of 8th international symposium Multiple-Valued Logic, Rosemont, Ill., IEEE and ACM, pp. 251-256, New York

[13] Lowerre, B.T., and Reddy, R.D. (1980) The Harpy Speech Understanding System. In *Trends in Speech Recognition.* Lea, W.A. (Ed.) Englewood Cliffs, Prentice-Hall, New York.

[14] Rosenblom, P. (1987) Best First Search. In *Encyclopedia of Artificial Intelligence*, Shapiro, S. (Ed) Vol. 2, Wiley, New York.

[15] Nordin, J.P. , Banzhaf W.(1995c) Controlling an Autonomous Robot with Genetic Programming. In proceedings of: *1996 AAAI fall symposium on Genetic Programming*, Cambridge, USA.

[16] Nordin, J.P. ,F. Francone, Banzhaf W. (1995) Explicitly Defined Introns in Genetic Programming. In *Advances in Genetic Programming II*,(In press) Kim Kinnear, Peter Angeline (Eds.) , MIT Press USA.

[17] Nordin J.P. and Banzhaf W. (1995a) Complexity Compression and Evolution, in *Proceedings of Sixth International Conference of Genetic Algorithms, Pittsburgh, 1995*, L. Eshelman (ed.), Morgan Kaufmann, San Mateo, CA

[18] Boyer R.S., and Moore J.S. (1979) Proving Teorems about LISP-Functions, In *J.ACM*, Vol. 22, pp 129-144.

[19] Boyer R.S. and Yu Y. (1992) Automated Correctness Proofs of Machine Code Programs for a Commercial Microprocessor, In *Automated Deduction - CADE-11* Kapur D. (Ed), pp. 416-430.

[20] Kleene, S.C. (1950) Introduction to Metamathematics, Van Nostrand, New York.

[21] Nordin, J.P., Banzhaf W.(1995c) Controlling an Autonomous Robot with Genetic Programming. In proceedings of: *AAAI fall symposium of Genetic Programming*, Cambridge, USA.

[22] Knight L., Haynes T. (1994) A GP Theorem Prover, technical report CS 7213, University of Tulsa